Aalborg Universitet

AALBORG UNIVERSITY
DENMARK

Program Analysis as Model Checking

Olesen, Mads Chr.

Publication date:
2014

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

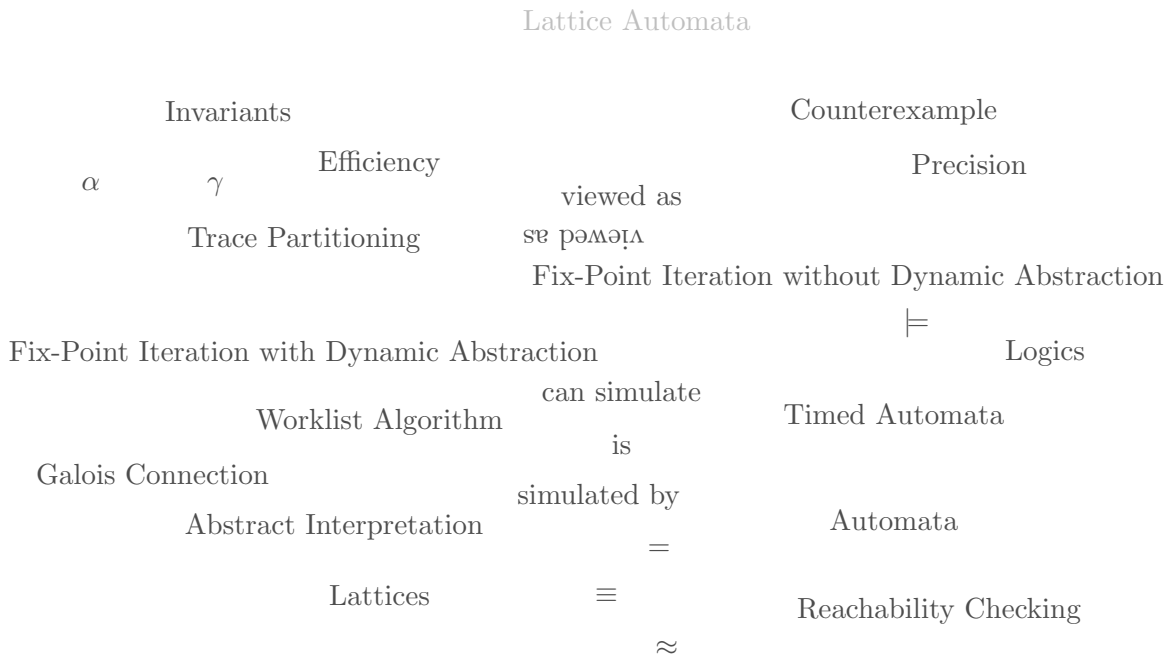
Citation for published version (APA):
Olesen, M. C. (2014). *Program Analysis as Model Checking*. Institut for Datalogi, Aalborg Universitet. Publication : Department of Computer Science, The Faculty of Engineering and Science, Aalborg University No. 85

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: May 04, 2023



Program Analysis as Model Checking

Mads Chr. Olesen

Department of Computer Science,
Aalborg University,
Denmark

PhD Thesis
Defended 20th December 2013

$$\gamma(\text{Jane}) = \top$$

Abstract

Software programs are proliferating throughout modern life, to a point where even the simplest appliances such as lightbulbs contain software, in addition to the software embedded in cars and airplanes. The correct functioning of these programs is therefore of the utmost importance, for the quality and sustenance of life. Due to the complexity inherent in the software it can be very difficult for the software developer to guarantee the absence of errors; automated support in the form of automated program analysis is therefore essential.

Two methods have traditionally been proposed: model checking and abstract interpretation. Model checking views the program as a finite automaton and tries to prove logical properties over the automaton model, or present a counter-example if not possible — with a focus on precision. Abstract interpretation translates the program semantics into abstract semantics efficiently represented as a lattice structure, and tries to use the invariants given by the lattice element to prove the absence of errors — with a focus on efficiency.

Previous work has argued that on a theoretical level the two methods are converging, and one method can indeed be used to solve the same problems as the other by a reformulation. This thesis argues that there is even a convergence on the practical level, and that a generalisation of the formalism of timed automata into lattice automata captures key aspects of both methods; indeed model checking timed automata can be formulated in terms of an abstract interpretation.

For the generalisation to lattice automata to have benefit it is important that efficient tools exist. This thesis presents multi-core tools for efficient and scalable reachability and Büchi emptiness checking of timed/lattice automata.

Finally, a number of case studies are considered, among others numerical analysis of C programs, and worst-case execution time analysis of ARM programs. It is shown how lattice automata allow automatic and manual tuning of the precision and efficiency of the verification procedure. In the case of worst-case execution time analysis a sound overapproximation of the hardware is needed; the case of identifying timing anomalous hardware for which such abstractions are hard to find is considered.

Dansk Sammenfatning

Softwareprogrammer findes overalt i det moderne liv, til et punkt hvor selv de simpleste apparater som lyspærer indeholder software, udover det software der er indlejret i biler og fly. Korrekt funktionalitet fra disse programmer er derfor af den yderste vigtighed, for at opretholde kvaliteten og i visse tilfælde tilstedeværelsen af liv. På grund af den kompleksitet der er medfødt i softwaren kan det være meget svært for software udvikleren at garantere fraværet af fejl; automatiseret support i form af automatiseret program analyse er derfor essentiel.

To metoder er traditionelt blevet foreslået: model checking og abstrakt fortolkning. Model checking ser et program som en endelig automat og forsøger at vise at en logisk egenskab holder, eller finde et modeksempel — med et fokus på præcision. Abstrakt fortolkning oversætter programmets semantik til en abstrakt semantik, effektivt repræsenteret i en lattice-struktur, og forsøger at bruge invarianter givet af lattice elementer til at bevise at fejl ikke kan forekomme — med et fokus på effektivitet.

Tidligere arbejder har argumenteret for at de to metoder konvergerer på et teoretisk plan og at en metode kan bruges til at løse problemer fra den anden ved en omformulering. Denne afhandling argumenterer for at denne konvergens også er sket på et praktisk plan, og at en generalisering af formalismen tids-automater til lattice-automater fanger nøgleaspekter fra begge metoder; model checking af tids-automater kan endda formuleres som en abstrakt fortolkning.

For at en sådan generalisering til lattice-automater har værdi, kræver det at effektive værktøjer eksisterer. Denne afhandling præsenterer multicore værktøjer der kan udføre effektiv og skalerbar reachability og Büchi tomheds-check for tids/lattice-automater.

Endeligt vil et antal eksempler på anvendelse blive gennemgået, bl.a. numerisk analyse af C programmer og værste kørselstid analyse af ARM programmer. Det vil blive vist hvordan lattice-automater kan bruges til automatisk og manuel tuning af præcision og effektivitet af verifikationsproceduren. I tilfældet for værste kørselstid analyse kræver det en abstrakt model der med garanti overapproksimerer hardwaren; der gives en definition af hvornår en given hardware udviser tids-anomaliteter, der gør det svært at finde en sådan abstrakt model.

Mandatory Page

Title: Program Analysis as Model Checking
Author: Mads Chr. Olesen
Supervisors:
Professor Kim Guldstrand Larsen,
Associate Professor René Rydhof Hansen

Published papers:

[42] Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Kenneth Yrke Jørgensen, Kim Guldstrand Larsen, Mads Chr. Olesen, Petur Olsen and Jiří Srba: opaal: A Lattice Model Checker. In *Proceedings of the International Symposium NASA Formal Methods (NFM)*, volume 6617 of *Lecture Notes in Computer Science*, pages 487–493. Springer, 2011.

[43] Andreas Engelbredt Dalsgaard, Alfons Laarman, Kim G. Larsen, Mads Chr. Olesen and Jaco van de Pol: Multi-Core Reachability for Timed Automata. In *Proceedings of Formal Modeling and Analysis of Timed Systems (FORMATS)* volume 7595 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2012.

[66] Alfons Laarman, Mads Chr. Olesen, Andreas Engelbredt Dalsgaard, Kim G. Larsen and Jaco van de Pol: Multi-Core Emptiness Checking of Timed Buchi Automata using Inclusion Abstraction In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 968–983. Springer, 2013.

[83] Mads Chr. Olesen, René Rydhof Hansen and Kim Guldstrand Larsen: An Automata-Based Approach to Trace Partitioned Abstract Interpretation. *Under submission.*

[31] Franck Cassez, René Rydhof Hansen and Mads Chr. Olesen: What is a Timing Anomaly?. In *Proceedings of the 12th International Workshop on Worst-Case Execution-Time Analysis*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the submitted or published scientific papers which are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis. As part of the assessment, co-author statements have been made available to the assessment committee and are also available at the Faculty. The thesis is not in its present form acceptable for open publication but only in limited and closed circulation as copyright may not be ensured.

Acknowledgments

I would like to thank my supervisors, Kim Guldstrand Larsen and René Rydhof Hansen, for their excellent guidance in the great adventure that is science. Not only have you been my supervisors, you have also become my friends; thanks for sharing coffee, beer, Wiener Melange, advice, anecdotes, long hours, crazy ideas and lessons learned.

Many thanks should also be given to my office-mates over the years: Martin Toft, Andreas Dalsgaard, Sebastian Biallas, Kenneth Jørgensen and Mikael Møller (even though I owe you one). An office space without people is just an empty room; thank you all for filling the empty room with your presence, humour and companionship in the journey that is a PhD study. Thanks to the members of "Madklubben": Line Juhl, Simon Laursen and Erik Wognsen; you learn a lot about someone when you share a meal together, and we have shared many meals. May the "rævesovs" be with you, always. A special thanks should go to Franck, Sebastian, Ralf, Thomas, Peter Höfner and Rob van Glabbeek for making my six months stay down under very pleasurable, and for being my lunch companions away from home.

The work in this thesis would not have been possible without my co-conspirators, that I have had the great privilege of working together with: Franck Cassez, Alfons Laarman, Jaco van de Pol, Petur Olsen, Jörg Brauer, Jiří Srba and Ralf Huuck. I hope our collaboration will continue.

Life is not only work; thank you to the F-klub. You can leave, but you can never check out. Thank you Bo, Markus, Thomas, Henrik, Jesper and all others who have shared a beer, a boardgame, and a weird discussion. When all else fails, it is nice to be able to go to Hal9k every Thursday, and spend an evening doing totally unrelated awesome stuff together with Anders, Alex, Mikael and all the other hackers.

Thank you far, mor, Anne, Signe, farmor and Anne for being my family, with all that goes with it, and supporting me in whatever crazy plans I come up with.

Finally, the biggest thanks should go to Jane. I know this journey has at times been at least as hard for you as it has been for me. Thanks for supporting me in this small adventure, part of the bigger adventure that we share.

Contents

Chapter 1

Introduction

The human civilisation is increasingly relying on automated machines, that for reasons of cost and flexibility are controlled by programmable computers; in our homes in the form of all sorts of appliances, on the roads in the form of self-driving vehicles, auto-piloted airplanes in the air above us, and even further away satellites that are crucial for the everyday communication around the globe that we have come to take for granted. Incorrect functioning of the computers we surround ourselves with can therefore have any number of consequences from a minor inconvenience, to injury or death. The verification of the correct functionality of programs is therefore of paramount importance.

The properties to be verified can range from simple properties like *"The program does not crash"* to more complicated properties like *"The program always produces the desired result"*, and in many cases also properties about timeliness like *"The program always responds within 5 seconds"*.

Several methods for verifying programs have been put forward over the years, allowing for different trade-offs between the level of trustworthiness and the precision, cost and automation of the analysis. All of them prove (in different ways) invariants about a program, that is, a logical formula that holds for any execution of the program.

Theorem Proving affords a very high level of verification, but at the cost of a massive effort. Verifying the seL4 microkernel of 10,000 lines of code took an effort of 20 personyears [64]. Such proofs typically build on a number of assumptions, e.g., that the hardware works as specified and that the compiler also works correctly.

The human-written proof can be verified automatically by a small verifier program, which is assumed to be correct. For additional trustworthiness several such verifiers can be written and run.

Abstract Interpretation is a method for analysing a program and extracting invariants about each point in the program. It works by ab-

stracting the concrete semantics of the programming language (which is assumed to be executed correctly) and extracting abstract semantics that are imprecise, but always over-approximating the concrete behaviour. The abstract semantics is typically defined in terms of an abstract domain, that captures the part of the semantics that is interesting for the verification at hand, e.g., the signedness or interval of variables. Abstract interpretation has been used to successfully verify industrial flight control software for the absence of runtime errors[1].

The design of a new abstract domain can require significant effort, but usage of that domain afterwards is automatic. Abstract interpretation typically prioritises efficiency, at the cost of precision. If an abstract interpretation is unable to find a strong enough invariant for verifying the correctness, a number of techniques can be applied at additional cost, such as trace partitioning or disjunctive completion.

Model Checking is an exhaustive method for verifying properties on a model of a system. The model is typically a form of automaton, of which the possible behaviours are explored looking for an error. The model is assumed to faithfully model the relevant behaviour of the real system, or in other words, the relation between the model and the real system needs to be proven[2]. Typically the behaviour of the model is finite, otherwise the naïve model checking degenerates into a semi-algorithm: if there is an error it will be found, but there is no guarantee of termination. Alternatively, a finite abstraction of the model can be used, as is done for model checking timed automata. Model checkers such as UPPAAL and SPIN have been successfully used to find errors in real-life programs and protocols[3].

For verifying programs the model can be automatically extracted from the program artifact itself. This process is typically done in an ad-hoc manner. It is worthwhile to notice that the model is itself an artifact that can be inspected, simulated and altered by a human. Simulating the execution of the model can, e.g., be used in debugging to replay a path. The model can be altered to, e.g., account for assumptions that can be made about the environment, or to exclude certain scenarios from the analysis.

The relative cost and verification depth has been plotted in Figure 1.1, for comparison. Besides cost and verification depth, another key aspect is traceability: whether the verification carried out has any connection to the object whose properties are to be verified. In abstract interpretation the traceability is typically ensured by the use of a Galois connection when

[1] Using Astreé [39].
[2] Such a proof could be done using methods from abstract interpretation.
[3] E.g., using UPPAAL [57] or Java PathFinder [97].

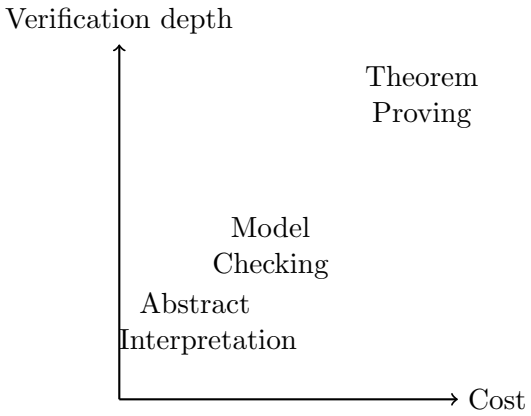


Figure 1.1: Various verification methods graphed in terms of typical cost and verification depth.

extracting the abstract semantics. In model checking the correspondence of the model to the real program is often simply assumed. However, if the standard language semantics do not apply (e.g., because of the possibility of hardware faults, or environment assumptions) the abstract semantics will have to be re-calculated for an abstract interpretation, where as such changes can easily be manually or automatically incorporated into the model for model checking.

The main focus of this dissertation is to use a combination of model checking and abstract interpretation to verify the correctness of programs. It will be shown that this combination allows abstraction techniques from abstract interpretation to effectively extract a model for verification with increased traceability, while techniques from model checking allow the verification to be carried out efficiently and with increased flexibility for the human overseeing the verification, in terms of ad hoc experiments or performance fine-tuning.

Chapter 2

Model Checking of Timed Automata Viewed as an Abstract Interpretation

In this chapter the modelling formalism timed automata [4] is introduced, and it will be shown how the model checking of such automata can be viewed as an abstract interpretation. Then the timed automata formalism will be generalised to lattice automata.

Timed automata build on finite automata, but are extended with a number of real-valued clock variables that increase at the same, constant rate — thereby allowing modelling of real-time systems in a convenient manner. For completeness, timed automata will be defined extended with discrete variables, and extended to networks of timed automata; this closely matches the formalism as implemented in UPPAAL [16, 72], but it should be noted that it does not add additional expressive power.

In Section 2.1 timed automata will be formally defined, whereafter the symbolic semantics used for timed automata model checking will be introduced in Section 2.2. In Section 2.3 the basic concepts of abstract interpretation will be introduced, and finally in Section 2.4 the connection between the two will be made, outlining how model checking timed automata can be viewed as an abstract interpretation. In Section 2.5 the model of lattice automata will be introduced, encompassing timed automata and the connection to program analysis briefly outlined.

2.1 Timed Automata

Timed automata are finite state machines with a finite set of real-valued, resettable clocks. Transitions between states can be guarded by constraints on clocks.

Definition 1 (Clock constraints)**.** *Let $\mathcal{G}(\mathcal{C})$ be the set of diagonal-free*[1] *clock constraints over the set of clocks $\mathcal{C}$, and let $g, g_1, g_2 \in \mathcal{G}(\mathcal{C})$:*

$$g := c \bowtie n \mid g_1 \wedge g_2$$

where $c \in \mathcal{C}$, $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, >, \geq, =\}$ is a comparison operator.

A clock constraint is *downwards closed* if it only imposes constraints on the upper bounds of clocks, i.e. only uses $<$ and $\leq$.

The time moments that a timed automaton can reach are defined as clock valuations.

Definition 2 (Clock valuation)**.** *A clock valuation for a set of clocks $\mathcal{C}$ is a function $v_{\mathcal{C}} : \mathcal{C} \to \mathbb{R}_{\geq 0}$.*

Two operations on clock valuations will be needed: $v_{\mathcal{C}} + d$ for delay s.t.

$$(v_{\mathcal{C}} + d)(c) = v_{\mathcal{C}}(c) + d$$

and reset of a set of clocks $r \subseteq \mathcal{C}$ s.t.

$$v_{\mathcal{C}}[r](c) = \begin{cases} 0 & \text{if } c \in r \\ v_{\mathcal{C}}(c) & \text{otherwise} \end{cases}$$

The notation $v_{\mathcal{C}} \models g$ means that the clock valuation $v_{\mathcal{C}}$ satisfies the clock constraint g.

Timed automata can be extended with discrete variables, a feature indispensable for modelling real-world systems. For this variable valuations needs to be defined.

Definition 3 (Variable valuation)**.** *For a finite set of integer variables V over a finite domain*

$$Dom(V) = \{n \in \mathbb{Z} | V_{min} \leq n \leq V_{max}\} \text{ for some } V_{min}, V_{max} \in \mathbb{Z}$$

a variable valuation is a function $v_V : V \to Dom(V)$.

Note that there are only finitely many variable valuations, for a given V and $Dom(V)$. The notation $Expr(V)$ is defined as expressions over the discrete variables, while

$$\begin{aligned} Stat(V) ::= \quad & v := Expr(V) \\ & \mid Stat(V); Stat(V) \end{aligned}$$

[1] To ensure the correctness of the later presented forward-reachability algorithm, guard constraints are not allowed to compare clocks, see Bouyer [26].

where $v \in V$, denotes statements assigning new values to the discrete variables, based on expressions over the discrete variables. For a variable valuation v_V and an expression $g \in Expr(V)$ let $v_V \models g$ mean that the variable valuation satisfies the expression. The variable valuation resulting from applying a statement $s \in Stat(V)$ to a valuation v_V is denoted as $v_V[s]$, with the usual semantics. If the set of variables V is empty the symbol "$\cdot$" will be used to denote the empty clock valuation.

Guard constraints over the clocks $\mathcal{C}$ and expressions over the discrete variables $Expr(V)$ are allowed to be mixed freely, to form *extended guard constraints*. In the following an extended guard g will be, in an abuse of notation, used both as a clock constraint and as an expression over the discrete variables.

Timed automata extended with variables can now be defined:

Definition 4 (Extended timed automaton)**.** *An extended timed automaton is a 7-tuple* $\mathcal{A} = (L, V, \mathcal{C}, Act, l_0, \rightarrow, I_{\mathcal{C}})$ *where*

- L *is a finite set of locations, typically denoted by* l
- V *is a finite set of integer variables over a finite domain* $Dom(V)$
- $\mathcal{C}$ *is a finite set of clocks, typically denoted by* c
- Act *is a finite set of actions*
- $l_0 \in L$ *is the initial location*
- $\rightarrow \subseteq L \times (\mathcal{G}(\mathcal{C}) \times Expr(V)) \times Act \times Stat(V) \times 2^{\mathcal{C}} \times L$ *is the (nondeterministic) transition relation, of which elements will be denoted as*

  $$l \xrightarrow{g,a,s,r} l'$$

  for a transition, where

  - l *is the source location,*
  - g *is an extended guard constraint over the clocks and discrete variables,*
  - a *is the action,*
  - s *is the update statement for the discrete variables,*
  - r *is the set of clocks reset, and*
  - l' *is the target location.*
- $I_{\mathcal{C}} : L \rightarrow \mathcal{G}(\mathcal{C})$ *is a function mapping locations to downwards closed clock constraints, giving an invariant for each location.*

An example of the graphical representation of an extended timed automaton is given in Figure 2.1. Locations are represented by nodes in the automaton with invariants written below the locations, transitions by edges in the automaton with guards and resets written above.

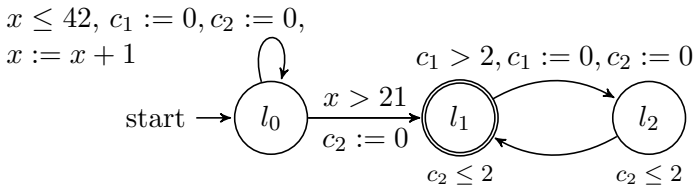


Figure 2.1: An example of an extended timed automaton, with clocks $\mathcal{C} = \{c_1, c_2\}$ and variables $V = \{x\}$. All transitions implicitly have the action τ, and the set of actions is $Act = \{\tau\}$.

2.1.1 Networks of Extended Timed Automata and Semantics

A network of timed automata is a parallel composition of extended timed automata that enables synchronisation over a finite set of channel names $Chan$. Let $ch!$ and $ch?$ denote the output and input action on a channel $ch \in Chan$.

Definition 5 (Network of timed automata)**.** *Let*

$$Act = \{ch!, ch?|ch \in Chan\} \cup \{\tau\}$$

be a finite set of actions, and let $\mathcal{C}$ be a finite set of clocks. Then the parallel composition of the extended timed automata

$$\mathcal{A}_i = (L_i, V, \mathcal{C}, Act, l_0^i, \rightarrow_i, I_{\mathcal{C}}^i)$$

for all $1 \leq i \leq n$, where $n \in \mathbb{N}$, is a network of timed automata, denoted

$$\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2||\ldots||\mathcal{A}_n$$

The concrete semantics of a network of timed automata is defined over the timed transition system $(S, \Rightarrow, Chan \cup \{\tau\} \cup \mathbb{R})$ s.t.

1. S is the set of states, where each element is of the form $(l_1, l_2, \ldots, l_n, v_V, v_{\mathcal{C}})$ where $l_i \in L_i$ are locations, v_V is a variable valuation and $v_{\mathcal{C}}$ is a clock valuation.

2. $\Rightarrow$ is the transition relation such that there are three types of transitions:

   - Discrete transitions:

$$(l_1, \ldots, l_i, \ldots, l_n, v_V, v_{\mathcal{C}}) \stackrel{\tau}{\Rightarrow} (l_1, \ldots, l_i', \ldots, l_n, v_V', v_{\mathcal{C}}')$$

if an edge

$$l_i \xrightarrow{g,\tau,s,r} l_i'$$

exists and guards

$$v_V \models g \text{ and } v_\mathcal{C} \models g$$

are satisfied as well as updated variables

$$v_V' = v_V[s] \text{ and clocks } v_\mathcal{C}' = v_\mathcal{C}[r]$$

satisfy invariants

$$v_\mathcal{C}' \models I_\mathcal{C}^i(l_i') \wedge \bigwedge_{k \neq i} I_\mathcal{C}^k(l_k)$$

- Binary synchronisation transitions:

  $(l_1, \ldots, l_i, \ldots, l_j, \ldots, l_n, v_V, v_\mathcal{C}) \overset{a}{\Rightarrow} (l_1, \ldots, l_i', \ldots, l_j', \ldots, l_n, v_V', v_\mathcal{C}')$

  if edges

  $$l_i \xrightarrow{g_i,a!,s_i,r_i} l_i' \text{ and } l_j \xrightarrow{g_j,a?,s_j,r_j} l_j'$$

  exists and guards

  $$v_V \models\ g_i \wedge g_j \text{ and } v_\mathcal{C} \models g_i \wedge g_j$$

  are satisfied as well as updated variables

  $$v_V' = (v_V[s_i])[s_j] \text{ and clocks } v_\mathcal{C}' = v_\mathcal{C}[r_i \cup r_j]$$

  satisfy invariants

  $$v_\mathcal{C}' \models I_\mathcal{C}^i(l_i') \wedge I_\mathcal{C}^j(l_j') \wedge \bigwedge_{k \notin \{i,j\}} I_\mathcal{C}^k(l_k)$$

- Delay transitions, by d time units:

  $$(l_1, \ldots, l_n, v_V, v_\mathcal{C}) \overset{d}{\Rightarrow} (l_1, \ldots, l_n, v_V, v_\mathcal{C} + d)$$

  for $d \in \mathbb{R}_{\geq 0}$ if

  $$v_\mathcal{C} + d \models \bigwedge_{k=1}^{n} I_\mathcal{C}^k(l_k)$$

As noted previously, modelling using a network of timed automata does not add expressive power over modelling with just one timed automaton, since any network of timed automata can be flattened to a single timed automaton. An example of a network of timed automata is given in Figure 2.2. The two automata cannot synchronise on the first transition unless both guards are fulfilled, and one automaton cannot take the transition without the other. Thus, a valid run of the network is:

$$\begin{aligned}&((l_0, l_2), \cdot, [c_1 = 0]) \overset{5}{\Rightarrow} ((l_0, l_2), \cdot, [c_1 = 5]) \overset{0.1}{\Longrightarrow} ((l_0, l_2), \cdot, [c_1 = 5.1]) \overset{a}{\Rightarrow} \\ &((l_1, l_3), \cdot, [c_1 = 5.1]) \overset{\tau}{\Rightarrow} ((l_1, l_4), \cdot, [c_1 = 0])\end{aligned} \tag{2.1}$$

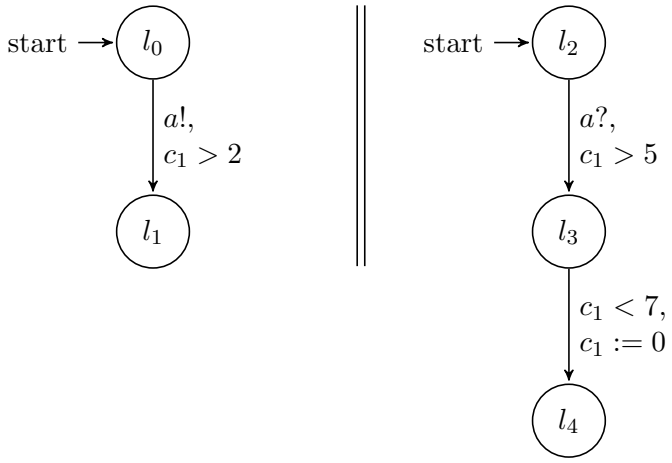


Figure 2.2: An example of a network of two timed automata $\{A_1, A_2\}$, with clocks $\mathcal{C} = \{c_1\}$, and the set of actions is $Act = \{\tau, a!, a?\}$.

2.2 Symbolic Semantics for a Timed Automaton

The concrete semantics of timed automata gives rise to an uncountable state space [16]. To model check it a finite abstraction of the state space is needed; the abstraction used by most timed automata model checkers is the zone abstraction [27]. Zones are sets of clock constraints that can be efficiently represented by Difference Bounded Matrices (DBMs) [19].

Definition 6 (Zones)**.** *Similar to the clock constraints of Definition 1, let $\mathcal{Z}(\mathcal{C})$ be the set of clock constraints over the set of clocks $c, c_1, c_2 \in \mathcal{C}$ generalized by:*

$$\mathcal{Z}(\mathcal{C}) ::= c \bowtie n \mid c_1 - c_2 \bowtie n \mid \mathcal{Z}(\mathcal{C}) \wedge \mathcal{Z}(\mathcal{C}) \mid \mathit{true} \mid \mathit{false}$$

where $n \in \mathbb{Z}$ is a constant, and $\bowtie \in \{<, \leq, >, \geq\}$ is a comparison operator. The equals operator, $=$, will be used as short-hand for the conjunction of $\leq$ and $\geq$.

This set of clock constraints are known as zones*, where a zone is typically denoted by Z.*

Zones represent (infinite) sets of clock valuations. The notation $v_\mathcal{C} \models Z$ means that the clock valuation $v_\mathcal{C}$ is included in the zone Z, and for the set of clock valuations in a zone the notation $[\![Z]\!] = \{v_\mathcal{C} \mid v_\mathcal{C} \models Z\}$ will be used.

The fundamental operations on zones are, for two zones Z and Z':

- $Z \uparrow$ modifying the constraints such that the zone represents all the clock valuations that can result from a delay from the current constraint set:

$$[\![Z \uparrow]\!] = \{v_\mathcal{C} + d | d \in \mathbb{R}_{\geq 0}, v_\mathcal{C} \in [\![Z]\!]\}$$

- $Z \cap Z'$ adding additional constraints to the zone Z, e.g. because a transition is taken that imposes a clock constraint (clock constraints can also be represented as a zone, which will be used later in a slight abuse of notation).

  $$[\![Z \cap Z']\!] = [\![Z]\!] \cap [\![Z']\!]$$

  The additional constraints might also make the zone empty, meaning that no clock valuations can satisfy the constraints.

- $Z[r]$ where $r \subseteq C$ is a clock reset of the clocks in r:

  $$[\![Z[r]]\!] = \{v_{\mathcal{C}}[r] | v_{\mathcal{C}} \in [\![Z]\!]\}$$

- $Z \subseteq Z'$ for checking if the constraints of Z' imply the constraints of Z, i.e. Z' is a less constrained zone, or equivalently: Z' contain the clock valuations of Z and possibly more:

  $$Z \subseteq Z' \text{ iff } [\![Z]\!] \subseteq [\![Z']\!]$$

Zones are usually represented using Difference Bound Matrices (DBMs)[47, 19], which will be introduced later in Definition 19.

Definition 7 (Zone semantics)**.** *The semantics of an extended timed automata*

$$\mathcal{A} = (L, V, \mathcal{C}, Act, l_0, \rightarrow, I_{\mathcal{C}})$$

under the zone abstraction is a simulation graph:

$$SG(\mathcal{A}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$$

such that:

1. *$\mathcal{S}_{\mathcal{Z}}$ consists of triples (l, v_V, Z) where $l \in L$ is a location, v_V is a variable valuation, and $Z \in \mathcal{Z}(\mathcal{C})$ is a zone.*

2. *$s_0 \in \mathcal{S}_{\mathcal{Z}}$ is the initial state $(l_0, v_V^0, Z_0 \uparrow \wedge\ I_{\mathcal{C}}(l_0))$ with $v_V^0(v) = 0$ and $Z_0 = \bigwedge_{c \in \mathcal{C}}(c = 0)$.*

3. *$\mathcal{T}_{\mathcal{Z}}$ is the symbolic transition relation using zones, s.t. $(s, s') \in \mathcal{T}_{\mathcal{Z}}$, denoted $s \Rightarrow s'$ with $s = (l, v_V, Z)$ and $s' = (l', v'_V, Z')$, if an edge*

   $$l \xrightarrow{g,a,s,r} l'$$

   exists, and $Z \wedge g \neq$ false, and $v_V \models g$, and

   $$v'_V = v_V[s]$$

   $$Z' = (((Z \wedge g)[r]) \uparrow) \wedge\ I_{\mathcal{C}}(l')$$

   and $Z' \neq$ false.

In the symbolic semantics the run illustrated in (2.1) on page 8 would be:

$$((l_0, l_2), \cdot, [c_1 \geq 0]) \overset{a}{\Rightarrow} ((l_1, l_3), \cdot, [c_1 > 5]) \overset{\tau}{\Rightarrow} ((l_1, l_4), \cdot, [c_1 \geq 0]) \tag{2.2}$$

Note how there are no delay transitions, as these are implicitly applied after the discrete transitions.

Extrapolation. Extrapolation with respect to maximal bounds [14, 4] is needed to make the symbolic state space finite. Basically, it is a mapping for each clock indicating the maximal possible constant the clock can be compared to in the future. It is used in such a way that if the value of a clock has passed its maximal constant, the clock's value is indistinguishable for the model, and can be set to ∞ to no longer track the precise value.

As an example consider the timed automaton in Figure 2.3, with regards to the clock c_2: any two clock valuations $v_{\mathcal{C}}, v'_{\mathcal{C}}$ such that

$$\forall c \in \mathcal{C} \setminus \{c_2\} : v_{\mathcal{C}}(c) = v'_{\mathcal{C}}(c)$$

and

$$v_{\mathcal{C}}(c_2) > 42 \wedge v'_{\mathcal{C}}(c_2) > 42$$

are indistinguishable for the automaton.

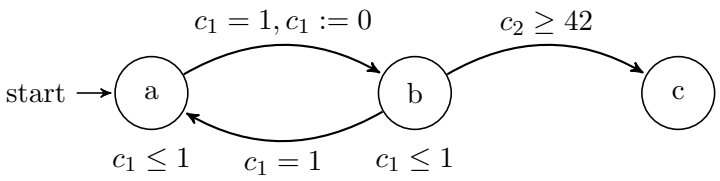


Figure 2.3: Timed automaton with maximal constant 42.

The extrapolation operation on zones is defined as:

- Z/B doing maximal bounds extrapolation, where $B : \mathcal{C} \to \mathbb{N}_0$ is the maximal bounds needed to be tracked for each clock.

Given a maximal bounds function B, it can be incorporated into the symbolic semantics such that

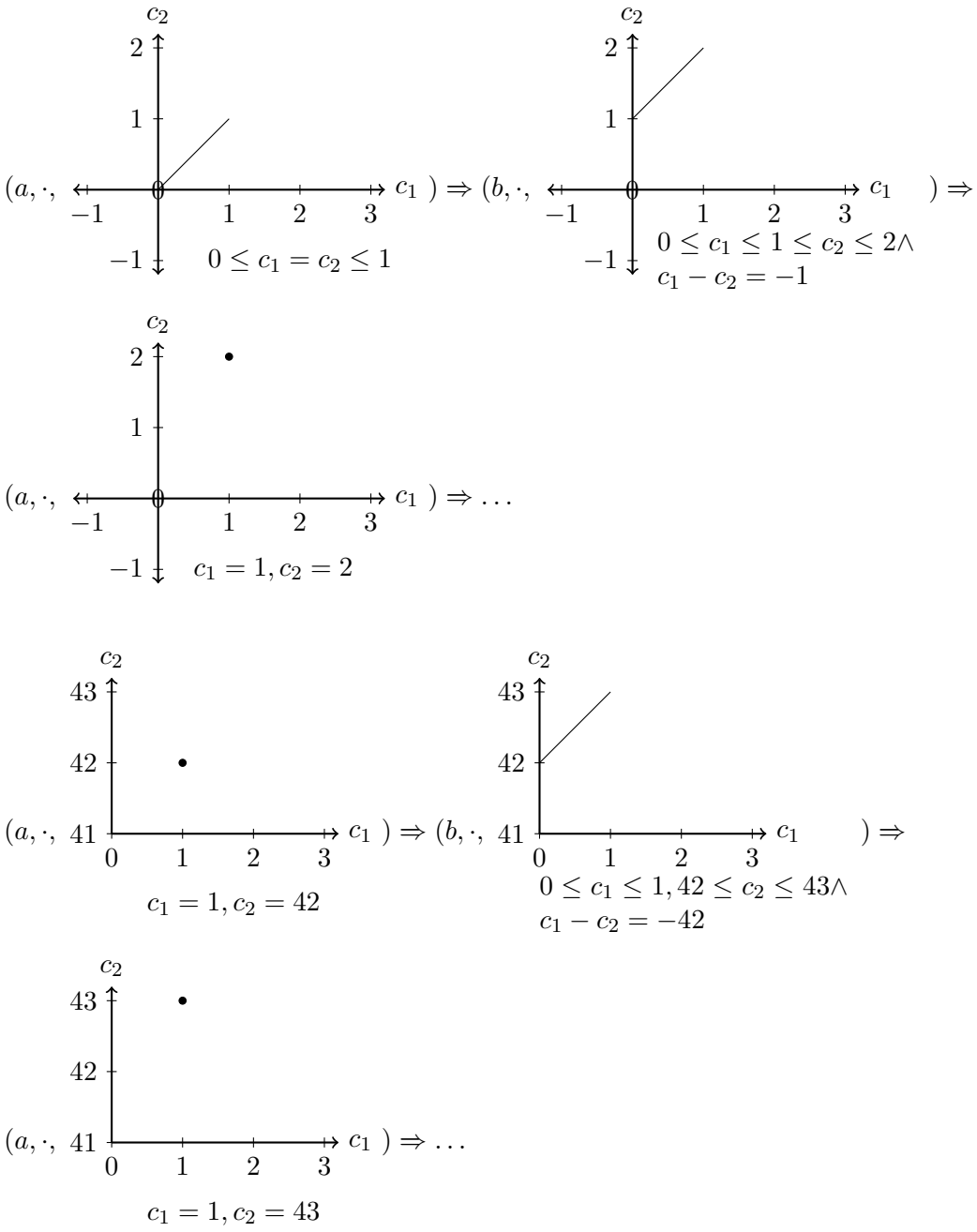
$$(l, v_V, Z) \Rightarrow (l', v'_V, Z')$$

becomes

$$(l, v_V, Z/B) \Rightarrow (l', v'_V, Z'/B),$$

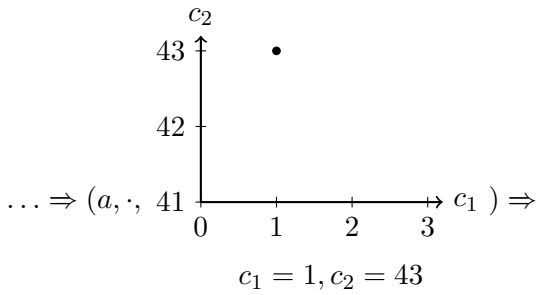
which in turn induces a finite transition system.

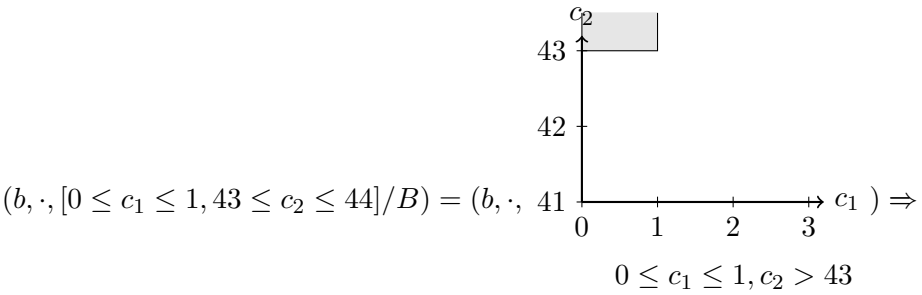
As an example, consider the un-extrapolated run of the timed automaton in Figure 2.3:

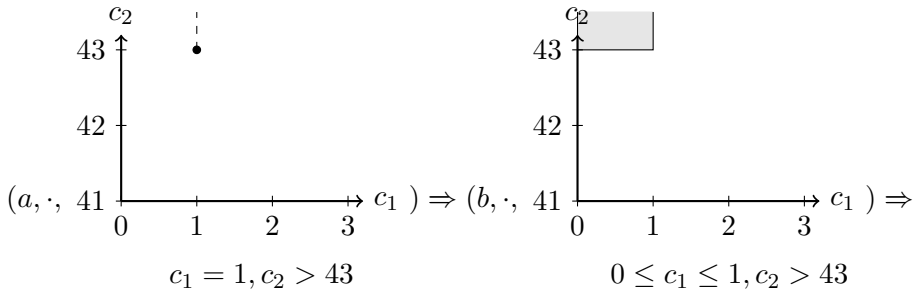


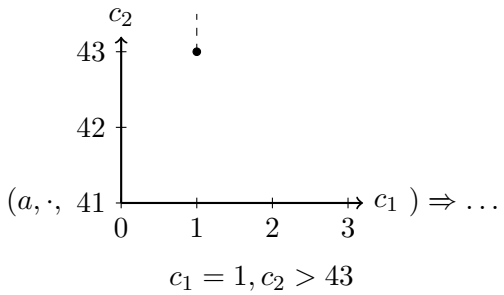
Clearly this run can be continued indefinitely without repeating a state, with the transition to the c location being enabled, from that point on,

With max clock bounds function $B(\cdot) = 42$ the same run in the extrapolated transition system becomes:









Where the state $(a, \cdot, [c_1 = 1, c_2 > 43])$ repeats, and thus the run can no longer be continued indefinitely without repeating a state.

The maximal clock bounds function B can be defined in a more or less precise manner: globally [5], per clock, per clock in each location [14] or

using both lower- and upper bounds [15]. It is typically derived by a coarse pre-analysis propagating clock bounds backwards along edges [14], but it could also be viewed as a simple static analysis in the monotone framework. Such a formulation will however not be pursued further, in this thesis.

Reachability Algorithm. The location reachability problem asks whether a certain location l_g is reachable, that is whether there exists some run ending in a state (l_g, v_V, Z) for some variable valuation v_V and zone Z. The location reachability problem is important, because any question about clock valuation reachability can be encoded as an location reachability problem.

Using the symbolic zone semantics under extrapolation, the location reachability algorithm for timed automata can be stated as Algorithm 1.

Algorithm 1 Location reachability for timed automata.

proc reachability(l_g)
  $W := \{$ INITIAL-STATE() $\}$; $P := \emptyset$
  while $W \neq \emptyset$
    $W := W \setminus (l, v_V, Z)$ **for some** $(l, v_V, Z) \in W$
    $P := P \cup \{(l, v_V, Z)\}$
    for (l', v'_V, Z') s.t. $(l, v_V, Z) \Rightarrow (l', v'_V, Z')$ **do**
      if $l' = l_g$ **then report & exit**
      if $\nexists Z''$: $(l', v'_V, Z'') \in W \cup P \wedge Z' \subseteq Z''$
        $W := W \setminus \{(l', v'_V, Z'') \mid Z'' \subseteq Z'\} \cup (l', v'_V, Z')$

The algorithm and the symbolic zone state space have here been stated without proof, or argumentation, that they are indeed correct, that is:

- Sound: if the algorithm reports a location is reachable, the location is.
- Complete: if a location is reachable the algorithm will report so.
- Terminating: the algorithm terminates for all inputs.

In the following the framework of abstract interpretation [34] will be introduced, whereafter the correctness will be argued for within that framework.

2.3 Abstract Interpretation

Abstract interpretation is concerned with deriving a set of invariants for each point in a program, by interpreting the semantics of the program abstractly in an abstract domain. Most commonly, the relation between the concrete semantics and the abstract semantics are specified using a *Galois connection*. Most commonly, an invariant before/after each syntactic statement of the

program is computed, although more precise invariants can be found using *trace partitioning*; this will be explored in Chapter 7.

Abstract interpretation uses abstract domains to efficiently represent program invariants. An abstract domain is at least a join semi-lattice, as defined by a partial order operator and a least upper bound, but often also a meet operator is defined making it a complete lattice.

Definition 8 (Complete Lattice)**.** *A complete lattice is a* 6*-tuple*

$$\mathcal{L} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$$

where

- D *is a set,*
- $\sqsubseteq$: $D \times D$ *is a partial order, i.e. reflexive, transitive and anti-symmetric.*
- $\sqcup : D \times D \to D$ *is a least upper bound operator s.t. it is an upper bound operator:*

  $$\text{if } a \sqcup b = c \text{ then } a \sqsubseteq c \wedge b \sqsubseteq c$$

  and it is a least upper bound operator:

  $$\text{if } a \sqcup b = c \text{ then } \forall d \in D : a \sqsubseteq d \wedge b \sqsubseteq d \implies d = c \vee c \sqsubseteq d$$

- $\sqcap : D \times D \to D$ *is a greatest lower bound operator s.t. it is a lower bound operator:*

  $$\text{if } a \sqcap b = c \text{ then } c \sqsubseteq a \wedge c \sqsubseteq b$$

  and it is a greatest lower bound operator:

  $$\text{if } a \sqcap b = c \text{ then } \forall d \in D : d \sqsubseteq a \wedge d \sqsubseteq b \implies d = c \vee d \sqsubseteq c$$

- $\bot$ *is a least element, i.e.* $\forall a \in D : \bot \sqsubseteq a$.
- $\top$ *is a greatest element, i.e.* $\forall a \in D : a \sqsubseteq \top$.

The lattice $\mathcal{L}$ will sometimes be used to refer to the underlying set D, when unambiguous.

2.3.1 Programs and Galois Connection

Definition 9 (Program)**.** *A* program *is taken to be a transition system* $(\mathcal{S}, Act, \rightarrow, s_0)$ *where* $\mathcal{S}$ *is the set of states, Act is the set of actions (statements),* $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ *is the transition relation, and* s_0 *is the initial state.*

Following convention, we write $s \xrightarrow{a} s'$ for the transition $(s, a, s') \in \rightarrow$.

Definition 10 (Traces of programs)**.** *A finite* trace *over a program is a finite sequence of states: $\sigma = s_0 \dots s_n$, such that for $0 \leq i \leq n$, $s_i \in \mathcal{S}$ and $s_i \xrightarrow{a} s_{i+1}$ for some $a \in Act$.*

We denote the final state of a trace σ as $\sigma_{\dashv}$. The set of all (finite) traces of a program P is denoted $[\![P]\!] = \{\sigma \in \mathcal{S}^* | \sigma \text{ is a finite trace of } P\}$ where $\mathcal{S}^*$ is the set of all sequences of states in $\mathcal{S}$.

In "standard" abstract interpretation, i.e., non-trace partitioning abstract interpretation, safety properties for a given program can be verified using approximations of the set of states that are reachable by the program. The sets of reachable states are usually represented using an abstract domain, D, with a concomitant concretisation function. For presentation purposes (and in preparation for future developments in Chapter 7) the powerset domain of program traces will be used as the concrete domain, although Galois connections in general can be between any lattices.

Definition 11 (Concretisation function)**.** *A* concretisation function

$$\gamma : D \to 2^{\mathcal{S}^*}$$

maps an abstract state to a super-set of traces whose states are all represented by the abstract state.

Note that $[\![P]\!] \subseteq 2^{\mathcal{S}^*}$, so the concretisation function might include traces not allowed by the program.

Given an abstraction function $\alpha : 2^{\mathcal{S}^*} \to D$ exists, a *Galois connection* can be formed.

Definition 12 (Galois connection)**.** *The functions:*

$$\alpha : 2^{\mathcal{S}^*} \to D \qquad \gamma : D \to 2^{\mathcal{S}^*}$$

form a Galois connection *iff [82, p. 236]*

$$\alpha(X) \sqsubseteq \ell \iff X \sqsubseteq \gamma(\ell)$$

as illustrated in Figure 2.4.

Note that if the functions α, γ form a Galois connection, they also uniquely determine each other [82, p. 239].

If α and γ, in addition to being a *Galois connection*, satisfy $\alpha \circ \gamma = id$ then they form a *Galois insertion*. Galois insertions (for the purposes of program analysis) have the nice property that no precision is lost by doing a concretisation and then an abstraction.

A *Galois connection*, comprising α and γ, can be used to induce an abstract model [92] of a program $P = (\mathcal{S}, Act, \rightarrow, s_0)$, by defining for each

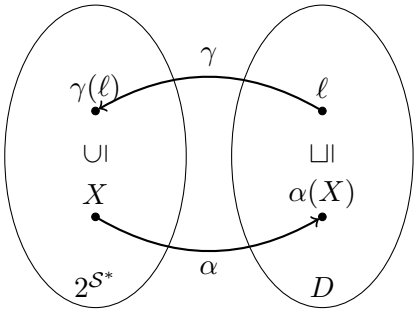


Figure 2.4: An illustration of the Galois connection defined in Definition 12, [82, Figure 4.6].

concrete action a corresponding abstract action, i.e. $a \in Act$ $f_a : D \to D$, that safely approximates the concrete semantics by requiring that for all $s, s' \in \mathcal{S}$ the following holds

$$s \xrightarrow{a} s' \text{ and } s_1 s_2 \cdots s \in X \text{ and } \alpha(X) = \ell \implies f_a(\ell) = \ell' \text{ and } s_1 s_2 \cdots s s' \in \gamma(\ell')$$

For any program, P, this *abstract model* of (all the actions of) a program is denoted: $\mathcal{M}_P = \{f_a | a \in Act\}$.

2.3.2 Computing a fix-point

For a program $P = (\mathcal{S}, Act, \to, s_0)$, an abstract domain $\mathcal{L} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, and an associated *abstract model* of a program $\mathcal{M}_P = \{f_a | a \in Act\}$, the strongest invariant expressible in the domain $\mathcal{L}$ can be computed for each *program location*, as the least fixpoint of a set of equations (see below). The set of program locations is denoted L, and a *program location* is the syntactical location in the program, e.g. the control-flow location. It is typically derived by splitting the set of program states $\mathcal{S}$ into $L \times MEM$ where L is the control flow locations, and MEM is the memory state i.e. values of variables. The abstract transformers f_a can be suitably split as well, such that

$$f_a : L \times D \to L \times D$$

The set of equations the fix-point needs to be computed over are given as a function $P : L \to D$:

$$P(l) = \bigsqcup \{\ell | l' \xrightarrow{a} l \text{ and } f_a(l', P(l')) = (l, \ell)\}$$

The least fix-point can then be computed using e.g. the worklist algorithm, as specified in Algorithm 2.

Algorithm 2 The standard worklist algorithm, used in abstract interpretation to compute a fix-point [82].

```
1   proc worklist(l_0, ℓ_0)
2   W ∈ 2^L ,  P : L → D
3   W := {l_0}
4   P(·,·) := ⊥, P(l_0) := ℓ_0
5
6   while W ≠ ∅
7     W := W \ {l}  for some l ∈ W
8     for (l', ℓ') s.t. l -a-> l' and f_a(l, P(l)) = (l', ℓ')  do
9       if ℓ' ⋢ P(l')  then
10        P(l') := P(l') ⊔ ℓ'
11        W := W ∪ {l'}
```

2.3.3 Widening for Faster Convergence

For an abstract domain $\mathcal{L} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, it might be the case that this fix-point computation does not converge. It could be that $|D| = \infty$ and there are infinitely ascending chains, i.e. an infinite sequence

$$\ell_0 \sqsubseteq \ell_1 \sqsubseteq \dots$$

or it might be the case that the fix-point computation simply converges too slowly to be useful in practice. A method for accelerating convergence is to use a *widening* operator.

Definition 13 (Widening operator)**.** *Given an abstract domain*

$$\mathcal{L} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$$

a function

$$\nabla : D \times D \to D$$

typically written inline as $\ell_1 \nabla \ell_2 = \nabla(\ell_1, \ell_2)$, is a widening operator iff:

- *For $\ell_1, \ell_2 \in D$ it holds that $\ell_1 \sqsubseteq \ell_1 \nabla \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \nabla \ell_2$*
- *For all ascending chains*

  $$\ell_0 \sqsubseteq \ell_1 \sqsubseteq \ell_2 \sqsubseteq \dots$$

  it holds that the ascending chain given by

  $$\ell_0 \sqsubseteq \ell_0 \nabla \ell_1 \sqsubseteq (\ell_0 \nabla \ell_1) \nabla \ell_2 \sqsubseteq \dots$$

  eventually stabilises.

A widening operator can be used to speed up the fix-point computation, but at the cost that the fix-point computed might not be the least fix-point. Widening can be introduced in Algorithm 2 simply by changing line 10 to:

$$P(l') \;:=\; P(l')\nabla\ell'$$

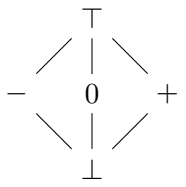
Because all ascending chains using ∇ eventually stabilise, convergence of this modified Algorithm 2 is guaranteed.

The fix-point reached by Algorithm 2 with widening can be improved by further iteration without widening, the reason being that widening might "overshoot" the least fix-point. Similarly, this iteration towards a fix-point can be accelerated using a *narrowing* operator, for which the details can be found in, e.g., [82].

2.3.4 Abstract Domains

A number of numerical domains typically used for program analysis will now be presented. It should be clear that they are each increasingly more precise: the sign domain, the interval domain, the difference bounded matrices domain and finally the octagon domain. In addition, a number of domain-theoretic constructs will be presented: the extension of a domain for a single variable to multiple variables, the disjunctive completion of a domain (turning it into a powerset domain), and the down-set completion of a domain.

The Sign Domain. The sign domain only keeps track of the sign of each variable. For a single variable the domain is shown in Figure 2.5.



$$\gamma(\top) = \{\ldots,-1,0,1,\ldots\}$$
$$\gamma(+) = \{1,\ldots\}$$
$$\gamma(0) = \{0\}$$
$$\gamma(-) = \{\ldots,-1\}$$
$$\gamma(\bot) = \emptyset$$

Figure 2.5: The Sign domain with the ordering shown as lines, and with associated concretisation function to the integers, $\gamma : D \to 2^{\mathbb{Z}}$.

Definition 14 (Sign domain for a single variable)**.** *The sign domain lattice is defined as* $\mathcal{L}_{\pm} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *where*

- D *is the set* $\{\bot, -, 0, +, \top\}$*,*
- $\sqsubseteq$*,* $\sqcup$*, and* $\sqcap$ *is given by Figure 2.5.*

- $\bot$ *is an "artificial" least element, i.e. it does not have a natural interpretation using the concretisation function.*
- $\top$ *is a greatest element, i.e. the concretisation function maps it to all concrete values.*

The concretisation function γ is as given in Figure 2.5.

For a set of variables the sign domain (and any numerical abstract domain in general) is extended in the natural way by the use of the Cartesian product, for which the partial order is applied component-wise.

Definition 15 (Extension to multiple variables)**.** *An abstract domain for a single variable, $\mathcal{L} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, with concretisation function $\gamma : D \to 2^{\mathbb{Z}}$, can be extended to multiple variables in the natural way by repeated application of the following definition extending to two variables:*

$$\mathcal{L}^2 = (D \times D, \sqsubseteq^2, \sqcup^2, \sqcap^2, (\bot, \bot), (\top, \top))$$

where

- $(a, b) \sqsubseteq^2 (c, d)$ *iff* $a \sqsubseteq c$ *and* $b \sqsubseteq d$.
- $(a, b) \sqcup^2 (c, d) = (e, f)$ *s.t.* $e = a \sqcup c$ *and* $f = b \sqcup d$.
- $(a, b) \sqcap^2 (c, d) = (e, f)$ *s.t.* $e = a \sqcap c$ *and* $f = b \sqcap d$.

and concretisation function $\gamma^2 : D \times D \to 2^{\mathbb{Z}} \times 2^{\mathbb{Z}}$, s.t.

$$\gamma^2(\ell, \ell') = (\gamma(\ell), \gamma(\ell'))$$

A graphical example of the type of invariants the Sign domain, extended to multiple variables, can represent is given in Figure 2.6.

The Interval Domain. The interval domain stores a lower- and an upper-bound for each variable. It is strictly more precise than the sign domain, at the expense of having an infinite cardinality, and even infinite ascending chains.

Definition 16 (Interval Domain)**.** *The interval domain lattice (illustrated in Figure 2.7) is defined as $\mathcal{L}_{interval} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ where*

- *D is the set $\{\bot\} \cup \{(x, y) \in (\mathbb{Z} \cup \{-\infty, \infty\}) \times (\mathbb{Z} \cup \{-\infty, \infty\}) | x \leq y\}$, that is all pairs (x, y) where $x \leq y$ where $\leq$ is extended to handle $-\infty$ and ∞ in the natural way,*
- *$\sqsubseteq$ is given by $\bot \sqsubseteq \ell$, $\ell \sqsubseteq \top$ and $[x, y] \sqsubseteq [x', y']$ iff $x' \leq x \wedge y' \geq y$*
- *$\sqcup$ is given by $\ell \sqcup \bot = \ell$, $\ell \sqcup \top = \top$, and $[x, y] \sqcup [x', y'] = [min(x, x'), max(y, y')]$*

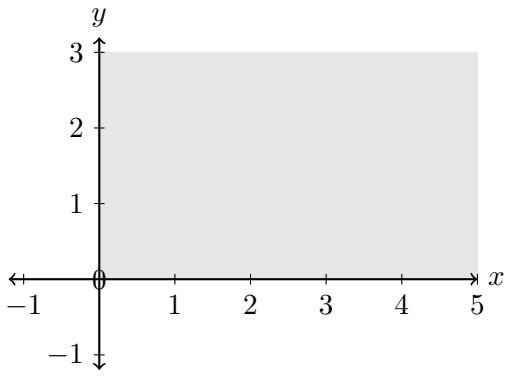


Figure 2.6: A representation of an element of the Sign domain for 2 variables, capturing the invariants $x \geq 0 \wedge y \geq 0$.

- $\sqcap$ *is given by* $\ell \sqcap \bot = \bot$, $\ell \sqcap \top = \ell$, *and* $[x, y] \sqcup [x', y'] = [max(x, x'), min(y, y')]$
- $\bot$ *is an "artificial" least element, i.e. it does not have a natural interpretation using the concretisation function.*
- $\top = [-\infty, \infty]$ *is the greatest element, i.e. the concretisation function maps it to all concrete values.*

The concretisation function γ *is as given by*

$$\begin{aligned} \gamma(\bot) &= \emptyset \\ \gamma(\top) &= \mathbb{Z} \\ \gamma([x, y]) &= \{i | x \leq i \leq y\} \end{aligned}$$

A graphical example of the type of invariants the Interval domain can represent is given in Figure 2.8.

Disjunctive Completion [35] (Powerset domain). The disjunctive completion of an underlying domain is a way of retaining precision that would otherwise be lost under join operations. It works by lifting the original domain from D to the powerset 2^D, and letting join add elements to the set. In this way a domain can gain power, by being able to express disjunctions, i.e. $x = 2 \vee x = 4$, or more powerful disjunctive invariants depending on the underlying domain.

Definition 17 (Disjunctive Completion [35])**.** *The disjunctive completion of a domain* $\mathcal{L} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *is defined as* $\mathcal{L}_{dis} = (2^D, \sqsubseteq', \cup, \cap, \emptyset, D)$ *where*

- $\sqsubseteq'$: $2^D \times 2^D$ *is defined as* $A \sqsubseteq' B$ *iff* $\forall \ell \in A. \exists \ell' \in B : \ell \sqsubseteq \ell'$.

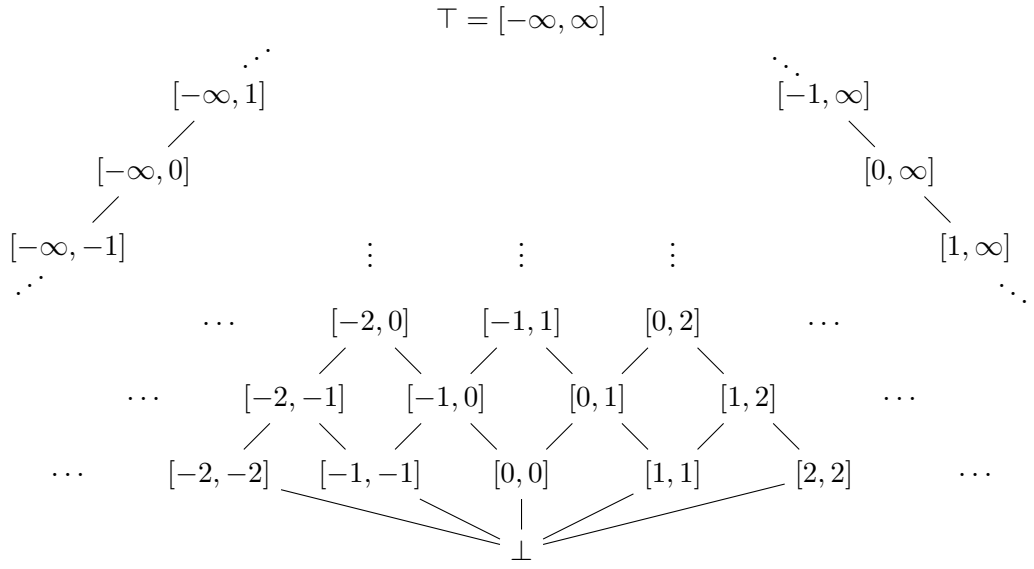


Figure 2.7: The interval domain for a single variable [82, Figure 4.2].

and the other operators are the regular operations from sets.

If the concretisation function for the domain $\mathcal{L}$ to a concrete domain A is given by $\gamma : D \to 2^A$, then the concretisation function for $\mathcal{L}_{dis}$ is given by $\gamma' : 2^D \to 2^A$, s.t.

$$\gamma'(\{\ell_0, \dots \ell_n\}) = \bigcup_{i=0}^{n} \gamma(\ell_i)$$

Note that Definition 17 allows for "redundancies" to remain in the lattice elements, e.g. if $\ell = \{a, b\}$ and $a \sqsubseteq b$, then because $\gamma'(\{a, b\}) = \gamma'(\{b\})$, a can be viewed as redundant. This leads to the definition of the *down closure* [36, Sec. 4.2.3.4] for a set X:

$$\downarrow X = \{y | \exists x \in X : y \sqsubseteq x\}$$

which allows the definition of the more effective, but equally precise, *down-set completion* [36, Sec. 4.2.3.4] (here following the presentation of [8]).

Definition 18 (Down-set Completion [36])**.** *The down-set completion of a domain $\mathcal{L} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is defined as $\mathcal{L}_{\downarrow} = (D', \sqsubseteq', \Omega \circ \cup, \cap, \emptyset, D)$ where*

- *$D' = \emptyset \cup \{X \in 2^D | \forall \ell, \ell' \in X : \ell \sqsubseteq \ell' \implies \ell = \ell'\}$ is the set of all non-redundant sub-sets of 2^D, i.e. the minimal sets. Note that for any $X \in 2^D$ there exists an $X' \in D'$ s.t. $\downarrow X = \downarrow X'$.*
- *$\sqsubseteq' : 2^D \times 2^D$ is defined as $A \sqsubseteq' B$ iff $\forall \ell \in A . \exists \ell' \in B : \ell \sqsubseteq \ell'$.*

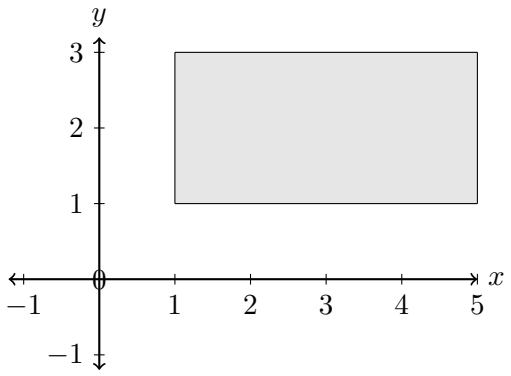


Figure 2.8: A representation of a element of the Interval domain for 2 variables, capturing the invariants $x \in [1,5] \wedge y \in [1,3]$.

- $\Omega : 2^D \to D'$ *is removing redundancies:*

$$\Omega(X) = X \setminus \{\ell \in X | \ell = \bot \vee \exists \ell' \in X . \ell \neq \ell' \wedge \ell \sqsubseteq \ell'\}$$

and the other operators are the regular operations from sets. The concretisation function is as given by Definition 17.

A graphical example of the type of invariants the disjunctive completion of the Interval domain can represent is given in Figure 2.9. In the following the disjunctive completion will be used to define derived domains, while the down-set completion readily allows these derived domains to be more efficient domain.

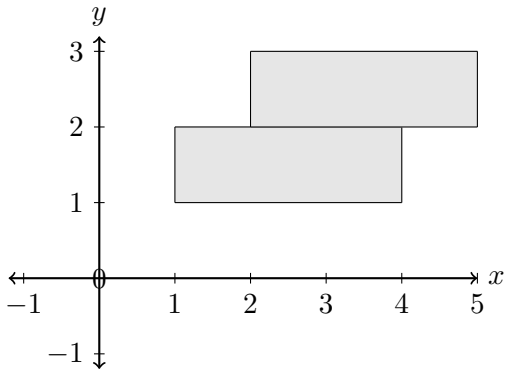


Figure 2.9: A representation of a element of the disjunctive completion of the Interval domain for 2 variables, capturing the invariants $(x \in [1,4] \wedge y \in [1,2]) \vee (x \in [2,5] \wedge y \in [2,3])$.

Difference Bound Matrices [47, 77, 19]. The DBM domain is a domain for representing not only the interval of a variable, i.e. $x \in [a, b]$, but also the interval of differences between variables, i.e. $x - y \in [a, b]$. They were originally intended for the verification of timed automata, for which they have been very successfully applied in e.g. UPPAAL [72]. Here, they are presented as an abstract domain, as was first done in [77].

For a set of variables V, the notation V_0 will denote $V \cup \{\mathbf{0}\}$, where $\mathbf{0}$ is a special variable that is always 0.

Definition 19 (Difference Bound Matrix Domain)**.** *The DBM domain lattice for a set of variables V is defined as $\mathcal{L}_{DBM} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ where*

- *D is the set of all matrices of size $(|V|+1)\times(|V|+1)$, where each matrix element is from $\mathbb{Z} \cup \{\infty\}$, and the natural ordering of $\mathbb{Z}$ is extended such that (in an abuse of notation) $\mathbb{Z} < \infty$. The i, j'th element of a DBM ℓ will be denoted as $\ell_{i,j}$.*

  Each matrix element represents a variable difference $x_i - x_j \leq c_{i,j}$, with differences against the special $\mathbf{0}$ variable expressing constraints of the form $x_i \leq c_{i,0}$ and $-x_i \leq c_{0,i}$. For presentation purposes all constraints will be non-strict, i.e. using $\leq$, as opposed to optionally being strict, i.e. $<$. Also, it will be assumed that all DBMs are in canonical form [19].

- *$\sqsubseteq$ is defined such that $\ell \sqsubseteq \ell'$ iff:*

$$\ell_{i,j} \leq \ell'_{i,j} \quad \forall 0 \leq i, j \leq |V| + 1$$

- *$\sqcup$ is defined such that $\ell = \ell' \sqcup \ell''$, gives each element of ℓ as:*

$$\ell_{i,j} = max(\ell'_{i,j}, \ell''_{i,j})$$

- *$\sqcap$ is defined such that $\ell = \ell' \sqcap \ell''$, gives each element of ℓ as:*

$$\ell_{i,j} = min(\ell'_{i,j}, \ell''_{i,j})$$

- *$\bot$ is an artificial least element, representing a DBM with unsatisfiable constraints. An unique element is used for this purpose.*

- *$\top$ is given as the unique element*

$$\top = \begin{bmatrix} 0 & \infty & \cdots & \infty \\ \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \cdots & 0 \end{bmatrix}$$

As the domain is a relational domain, the concretisation function must map DBMs to variable valuations in order for it to capture relational constraints precisely. The concretisation function $\gamma : \mathcal{L}_{DBM} \to (V \to \mathbb{Z})$ *is as given by*

$$\begin{aligned}\gamma(\ell) = \{v_V \in (V \to \mathbb{Z}) \mid \\ &\forall i, j.i \neq 0, j \neq 0, i \neq j : v_V(v_i) - v_V(v_j) \leq \ell_{i,j} \wedge \\ &\forall i : v_V(v_i) \leq \ell_{i,0} \wedge \\ &\forall i : -v_V(v_i) \leq \ell_{0,i}\}\end{aligned}$$

which consists of three conjunctions:

1. *Relational constraints between two variables*
2. *Upper bounds on variables, expressed as differences to the special* **0** *element*
3. *Lower bounds on variables, expressed as differences to the special* **0** *element*

A graphical example of the type of invariants a DBM can represent is given in Figure 2.10. The join operator is also known as the *convex*

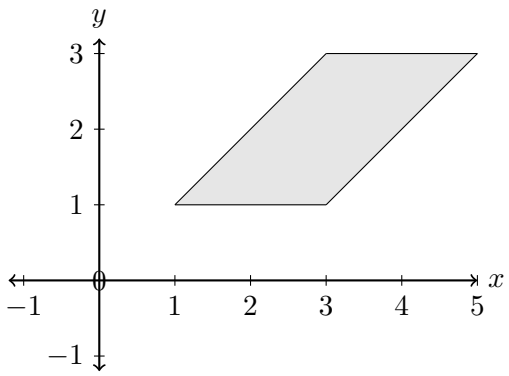


Figure 2.10: A representation of a DBM for 2 variables, capturing the invariants $x \in [1, 5] \wedge y \in [1, 3] \wedge x - y \in [0, 2]$.

hull operation, because it returns the smallest DBM that encompasses both operands. An example of the convex hull operation is shown in Figure 2.11. Note how for the DBMs a and b there an implicit invariant that $x - y \in [-1, 3]$, that is implied by the other constraints. This however becomes relevant when computing the convex hull, to e.g. eliminate the valuation $x = 5, y = 1$ that would have been included if the join was performed in the interval domain.

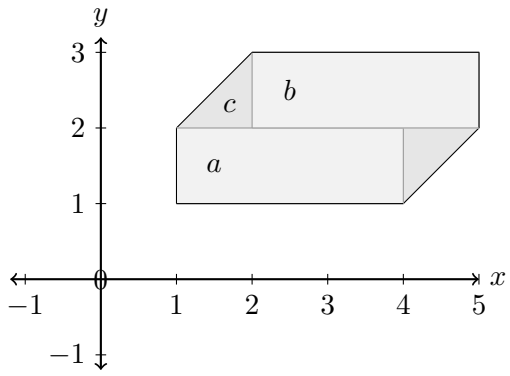


Figure 2.11: A representation of a DBM for 2 variables, showing the join of the two DBMs $a = (x \in [1,4] \wedge y \in [1,2])$ and $b = (x \in [2,5] \wedge y \in [2,3])$, and the result $c = (x \in [1,5] \wedge y \in [1,3] \wedge x - y \in [-1,3])$.

Octagons [78, 79]. Octagons are an extension of DBMs, such that not only the interval of a variable, i.e. $x \in [a,b]$, and the interval of differences between variables, i.e. $x - y \in [a,b]$, but the more general *sum constraints* [78] of the form $\pm x \pm y \leq a$ are tracked. For a set of variables $V = \{v_0, \ldots v_n\}$ a DBM over the twice as large set $V^+ = \{v_0^+, v_0^-, \ldots, v_n^+, v_n^-\}$ is constructed, as a representation of the octagon. Note that no special **0** variable needs to be introduced, as constraints of the form $v_i \leq a$ and $v_i \geq a$ can be represented by $v_i^+ - v_i^- \leq 2a$ and $v_i^- - v_i^+ \leq -2a$, respectively.

Definition 20 (Octagon Domain)**.** *The Octagon domain lattice for a set of variables V is defined as $\mathcal{L}_{OCT} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ where*

- *D is the set of all matrices of size $(|V| \cdot 2) \times (|V| \cdot 2)$, where each matrix element is from $\mathbb{Z} \cup \{\infty\}$, and the natural ordering of $\mathbb{Z}$ is extended such that (in an abuse of notation) $\mathbb{Z} < \infty$. The i,j'th element of an octagon ℓ will be denoted as $\ell_{i,j}$.*

  A row or column with index $2 \cdot i$ represents the variable v_i^+, and a row or column with index $2 \cdot i + 1$ represents the variable v_i^-. Thus, the matrix elements represents constraints of one of these forms:

  - *$\ell_{i,j}$ represents $v_i^+ - v_j^+ \leq a$*
  - *$\ell_{i,j+1}$ represents $v_i^+ - v_j^- \leq a$, i.e. $v_i + v_j \leq a$*
  - *$\ell_{i+1,j}$ represents $v_i^- - v_j^+ \leq a$, i.e. $-v_i - v_j \leq a$*
  - *$\ell_{i+1,j+1}$ represents $v_i^- - v_j^- \leq a$, i.e. $-v_i + v_j \leq a$*

  Interval constraints such as $v_i \leq a$ and $v_i \geq a$ are represented as $v_i^+ - v_i^- \leq 2 \cdot a$ and $v_i^- - v_i^+ \leq -2 \cdot a$ respectively.

Special care must be taken for the octagon to be in closed (canonical) form [78]. The closure algorithm depends on whether the concretisation function maps to integers, or to the reals or rationals; computing the smallest closure for the integers is more expensive than for the reals or rationals. In the following it will be assumed that all octagons are in closed form.

- $\sqsubseteq$ *is defined such that* $\ell \sqsubseteq \ell'$ *iff:*

$$\ell_{i,j} \leq \ell'_{i,j} \quad \forall 0 \leq i,j \leq |V| \cdot 2 - 1$$

- $\sqcup$ *is defined such that* $\ell = \ell' \sqcup \ell''$*, gives each element of* ℓ *as:*

$$\ell_{i,j} = max(\ell'_{i,j}, \ell''_{i,j})$$

- $\sqcap$ *is defined such that* $\ell = \ell' \sqcap \ell''$*, gives each element of* ℓ *as:*

$$\ell_{i,j} = min(\ell'_{i,j}, \ell''_{i,j})$$

- $\bot$ *is an artificial least element, representing an octagon with unsatisfiable constraints.*

- $\top$ *is given as the unique element*

$$\top = \begin{bmatrix} 0 & \infty & \cdots & \infty \\ \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \cdots & 0 \end{bmatrix}$$

Octagons are able to represent constraints that, in two dimensions, feature eight edges, hence the name. An example of an octagon representation is given in Figure 2.12.

More expressive domains for numerical analysis exist, such as polyhedra [41], but these will not be covered in detail in this thesis.

2.4 A Connection

In Section 2.2 the symbolic semantics for timed automata using zones represented as DBMs were reviewed. In Section 2.3 a number of abstract domains were presented, including the domain given by DBMs. In this section the model checking of a timed automaton will be cast as an abstract interpretation.

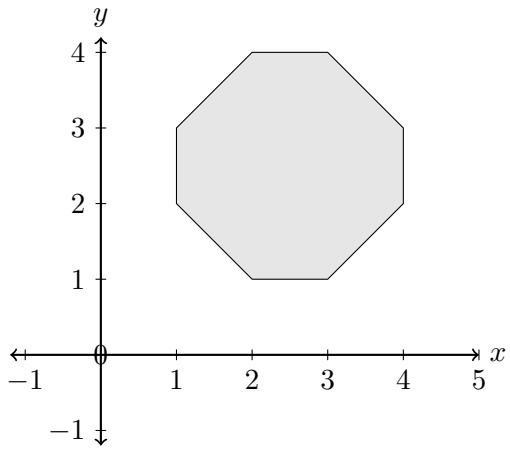


Figure 2.12: A graphical representation of an octagon for 2 variables, capturing the invariants $x \in [1,4] \wedge y \in [1,4] \wedge x - y \in [-2,2] \wedge x + y \in [3,7]$.

2.4.1 Domain Used for Model Checking Timed Automata

Timed automata model checking is a sound and complete procedure. The soundness can be directly established by a Galois connection between the concrete semantics, and the symbolic zone semantics. It should be noted that clocks in timed automata can readily be viewed as variables, with the exception of the special "delay" operation which increases all clock variables at the same rate. The notion of "clock" and "variable" will be used interchangeably in this section.

Galois Connection for Timed Automata Semantics. Any DBM from the DBM lattice (Definition 19) represents a convex set of clock valuations. The concretisation function maps any DBM to the set of clock valuations that satisfies its constraints:

$$\gamma(D) = \{v_{\mathcal{C}} | v_{\mathcal{C}} \models D\}$$

The abstraction function is straightforward, as any clock valuation describes a set of precise constraints on the clocks of the form $c_i = x_i$, starting from the unconstrained DBM $\top$. The only pitfall is that the clock valuation can involve real values, whereas a DBM can only involve constraints using integers; as such the abstraction needs to map the clock valuation to the smallest DBM including the clock valuation. This is given by the region

construction [5], and thus the abstraction function becomes:

$$\alpha(v_{\mathcal{C}}) = \top \sqcap \tag{2.3}$$

$$[c_0 \geq \lfloor v_{\mathcal{C}}(c_0) \rfloor \wedge c_0 \leq \lceil v_{\mathcal{C}}(c_0) \rceil)] \sqcap \ldots \sqcap [c_n \geq \lfloor v_{\mathcal{C}}(c_n) \rfloor \wedge c_n \leq \lceil v_{\mathcal{C}}(c_n) \rceil] \tag{2.4}$$

$$\sqcap \prod_{i=0,j=0}^{n} \begin{cases} [c_i - c_j = v_{\mathcal{C}}(c_i) - v_{\mathcal{C}}(c_j)] & \text{if } frac(v_{\mathcal{C}}(c_i)) = frac(v_{\mathcal{C}}(c_j)) \\ [c_i - c_j < \lfloor v_{\mathcal{C}}(c_i) \rfloor - \lfloor v_{\mathcal{C}}(c_j) \rfloor] & \text{if } frac(v_{\mathcal{C}}(c_i)) < frac(v_{\mathcal{C}}(c_j)) \\ [c_i - c_j > \lfloor v_{\mathcal{C}}(c_i) \rfloor - \lfloor v_{\mathcal{C}}(c_j) \rfloor] & \text{if } frac(v_{\mathcal{C}}(c_i)) > frac(v_{\mathcal{C}}(c_j)) \end{cases} \tag{2.5}$$

where $frac$ is the fractional part of the clock value. The big formula consists of (2.4) a number of interval constraints, and (2.5) a number of diagonal constraints.

Lemma 1. *α and γ as defined forms a Galois insertion from the set of clock valuations to DBMs.*

Using the domain of DBMs an abstract interpretation of the concrete semantics of a timed automaton

$$\mathcal{A} = (L, V, \mathcal{C}, Act, l_0, \rightarrow, I_{\mathcal{C}})$$

as given by Definition 4, can be constructed. The abstract semantics are defined over the timed transition system

$$(S, \Rightarrow, Act \cup \mathbb{R})$$

where

1. S consists of triples(l, v_V, D) where $l \in L$ is a location, v_V is a variable valuation and D is a DBM from the DBM domain.

2. $\Rightarrow$ is the transition relation s.t.

   - A discrete transition for an action $a \in Act$:
     $(l, v_V, D) \stackrel{a}{\Rightarrow} (l, v'_V, D')$ if an edge
     $l \xrightarrow{g,\tau,s,r} l'$ exists and $v_V \models g$ is satisfied, as well as updated variables $v'_V = v_V[s]$ and some clock valuation $v_{\mathcal{C}} \models D$ satisfies the guard $v_{\mathcal{C}} \models g$, and after performing the transition:

     $$D' = \alpha(\{v'_{\mathcal{C}} | v_{\mathcal{C}} \in \gamma(D) \text{ s.t. } v_{\mathcal{C}} \models g, v'_{\mathcal{C}} = v_{\mathcal{C}}[r], \text{ and } v'_{\mathcal{C}} \models I_{\mathcal{C}}(l)\})$$

     which can be re-written as

     $$D' = \alpha(\{v'_{\mathcal{C}} | v_{\mathcal{C}} \in \gamma(D \sqcap g) \text{ s.t. } v'_{\mathcal{C}} = v_{\mathcal{C}}[r], \text{ and } v'_{\mathcal{C}} \models I_{\mathcal{C}}(l)\})$$

(Because the clock constraint g can be viewed as a zone/DBM element)

$$D' = \alpha(\{v'_{\mathcal{C}} | v'_{\mathcal{C}} \in \gamma((D \sqcap g)[r]) \text{ s.t. } v'_{\mathcal{C}} \models I_{\mathcal{C}}(l)\})$$

(The clock reset can be performed in the abstract)

$$D' = \alpha(\{v'_{\mathcal{C}} | v'_{\mathcal{C}} \in \gamma((D \sqcap g)[r] \sqcap I_{\mathcal{C}}(l))\})$$

(The clock invariant can be viewed as a DBM element)

$$D' = \alpha(\gamma((D \sqcap g)[r] \sqcap I_{\mathcal{C}}(l)))$$

Finally, because α and γ form a Galois insertion, the successor DBM computation can be reduced to

$$D' = (D \sqcap g)[r] \sqcap I_{\mathcal{C}}(l)$$

- A delay by d time units:
  $(l, v_V, D) \overset{d}{\Rightarrow} (l, v_V, D')$ for $d \in \mathbb{R}_{\geq 0}$ if

$$D' = \alpha(\{v'_{\mathcal{C}} | v_{\mathcal{C}} \in \gamma(D) \text{ s.t. } v'_{\mathcal{C}} = v_{\mathcal{C}} + d \text{ and } v'_{\mathcal{C}} \models I_{\mathcal{C}}(l)\})$$

As delay transitions can be combined it is more efficient to compute the symbolic delay as the closed form of

$$\bigsqcup_{d=0\ldots\infty} \{D' | \alpha(\{v'_{\mathcal{C}} | v_{\mathcal{C}} \in \gamma(D) \text{ s.t. } v'_{\mathcal{C}} = v_{\mathcal{C}} + d \text{ and } v'_{\mathcal{C}} \models I_{\mathcal{C}}(l)\})\}$$

which corresponds to delaying between 0 and ∞ time units. For the case without an invariant (or $I_{\mathcal{C}}(l) = true$), this transformer exists and furthermore can be represented without loss of precision:

$$D' = D \uparrow$$

Concretely, this is implemented by removing all upper bounds from D. A valid concern is that an unbounded delay might allow a delay not possible in the concrete semantics due to an invariant. However, per Definition 4 invariants can only be downwards closed, meaning that if a delay of d is not allowed, then a delay of $d + x$ for any x is also not allowed.

Thus, an unbounded delay transition with an invariant $(l, v_V, D) \Rightarrow (l, v_V, D')$ can be represented as

$$D' = D \uparrow \sqcap I_{\mathcal{C}}(l)$$

As delay transitions are always enabled the semantics for a discrete transition followed by a delay transition can be combined into a single step. As delay transitions can be represented without loss of precision, this is often done for efficiency.

In the fixpoint computation, for each location a DBM representing the join of all clock valuations that the location can be reached by is kept. This is presented in Algorithm 3.

Algorithm 3 Abstract interpretation fixpoint computation for timed automaton with abstract semantics.

```
1   proc worklist(l_0, v_V^0, D_0)
2   W ∈ 2^{L×(V→Dom(V))} ,  P : L × (V → Dom(V)) → D
3   W := {(l_0, v_V^0)}
4   P(·,·) := ⊥, P(l_0, v_V^0) := D_0
5
6   while W ≠ ∅
7     W := W \ {(l, v_V)} for some (l, v_V) ∈ W
8     for (l', v'_V, D') s.t. (l, v_V, P(l, v_V)) ⇒ (l', v'_V, D') do
9       if D' ⋢ P(l', v'_V) then
10        P(l', v'_V) := P(l', v'_V) ⊔ D'
11        W := W ∪ {l', v'_V}
```

Lemma 2. *Algorithm 3 is sound, but not complete, with regards to location reachability.*

Proof. The soundness follows from the Galois connection. The incompleteness is due to the possible loss of precision of the join operator for the DBM, the convex-hull, that can result in "spurious" clock valuations being included in the state space, that can in turn result in transitions being spuriously enabled, and locations spuriously deemed reachable. □

Besides not being complete, another problem is that the DBM domain has infinite ascending chains, so the termination of Algorithm 3 is not guaranteed. The traditional method for ensuring termination, and the method for proving that model checking timed automata is decidable is the observation by Alur and Dill [5] that for a given timed automaton certain classes of clock valuations are indistinguishable – namely those for which the value of the clocks are above the maximal syntactic constant against which any clock is compared in the model. In the original work by Alur and Dill [5] the extrapolation was done using the maximal syntactic constant in the model, and using the same constant for all clocks.

The default solution is to use an "extrapolation" operator, enlarging DBMs that are indistinguishable.

Definition 21 (Global max constant extrapolation)**.** *Given a timed automaton $\mathcal{A}$ over the set of clocks $\mathcal{C}$, where the maximal constant appearing in any clock guard or invariant is k, the global max constant extrapolation*

is defined [14] $extra_k(D) = D'$ *s.t. for each element of the DBM* $D'_{i,j}$*:*

$$D'_{i,j} = \begin{cases} \infty, & \text{if } D_{i,j} > k. \\ -k, & \text{if } D_{i,j} < -k. \\ D_{i,j}, & \text{otherwise.} \end{cases}$$

Lemma 3. *Incorporating extrapolation in the abstract semantics, s.t. any transition* $(l, v_V, D) \Rightarrow (l', v'_V, D')$ *becomes* $(l, v_V, D) \Rightarrow (l', v'_V, extra_k(D'))$*, results in a finite transition system.*

Proof. The image of $extra_k$ is a finite set, immediately giving the result. □

Lemma 4. *For any reachable symbolic state* (l, v_V, D)*, and for every outgoing edge* $l \xrightarrow{g,a,s,r} l'$ *with clock constraint* $g \in \mathcal{G}(\mathcal{C})$ *it holds that*

$$D \sqcap g \neq false \iff extra_k(D) \sqcap g \neq false$$

meaning, that no edge clock guards will be enabled by applying an extrapolation [14].

2.4.2 Extrapolation: Abstraction or Widening?

Extrapolation can be viewed in one of two different ways: either as a model-specific abstraction operator for the Galois connection, or as a model-dependent widening operator. Both viewpoints will be explored.

Viewing extrapolation as an abstraction operator is straightforward, and follows the style of [45, 14, 15]. Using an extrapolation operator $extra : \mathcal{L}_{DBM} \to \mathcal{L}_{DBM}$, the abstraction of the Galois connection becomes $extra \circ \alpha$, while the concretisation operator remains the same. This viewpoint makes for a rather straight-forward analysis, but it does make the abstraction operator depend on the current model, which is unconventional in abstract interpretation.

A conventional mean of dealing with non-convergence in abstract interpretation is to use a widening. Widenings are typically instantiated per program, e.g. using all the constants in the program as widening points [34, 82]. Thus, for a more conventional analysis specification an extrapolation can be viewed as a widening.

Definition 22 (Extrapolation as Widening)**.** *An extrapolation operator*

$$extra : \mathcal{L}_{DBM} \to \mathcal{L}_{DBM}$$

can be turned into a widening operator (Definition 13) as:

$$D_0 \nabla_{extra} D_1 = extra(D_0) \sqcup extra(D_1)$$

2.4.3 Regaining Completeness

A key issue in (reachability) model checking of timed automata is that the procedure should be sound, terminating *and* complete. Using Algorithm 3 on an extrapolated transition system provides a sound and terminating procedure. To regain completeness the abstract domain needs to be altered slightly. The key observation is that the extrapolated transition system is finite; thus any imprecision is introduced by the joining. Lifting the DBM using the disjunctive completion makes the joining operator precise, regaining completeness.

For completeness the definition of the actual domain used is given below, although it is just the combination of using Definition 17 to lift Definition 19.

Definition 23 (Disjunctive Completion of the Difference Bound Matrix Domain)**.** *The disjunctive completion of the DBM domain lattice for a set of variables V given by $\mathcal{L}_{DBM} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, is defined as $\mathcal{L}_{DBM,dis} = (2^D, \sqsubseteq', \cup, \cap, \emptyset, \{\top\})$ where*

- *$\sqsubseteq'$: $2^D \times 2^D$ is defined as $A \sqsubseteq' B$ iff $\forall \ell \in A. \exists \ell' \in B : \ell \sqsubseteq \ell'$.*
- *The top element, $\{\top\}$ is given as the unique singleton set of $\top$ from $\mathcal{L}_{DBM}$.*

The concretisation function is as given in Definition 17.

Theorem 1. *Using Algorithm 3 with the domain from Definition 23 on an abstract semantics providing a finite transition system (such as Lemma 3) gives a sound, complete, and terminating procedure.*

Proof. The soundness is due to the Galois connection. The termination is due to the transition system being finite, by Lemma 3. Furthermore, the use of extrapolation does not enable any additional transitions by Lemma 4, thereby not affecting the completeness. The completeness is due to the use of a precise join operator, namely the set union of the disjunctive completion given by Definition 23. □

2.5 Lattice Automata

The formalism of timed automata can be generalised from using the DBM domain to using any lattice. This formalism will be called *lattice automata*.

Definition 24 (Lattice Automaton)**.** *A* lattice automaton *is a triple $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ where S is a finite set of locations, $\mathcal{L} = (D, \sqsubseteq, \sqcup)$ is a join semi-lattice and $\longrightarrow \subseteq S \times D \times S \times D$ is a transition relation which has the* monotonicity property*: for all $s_1, s_2 \in S$ and $\ell_1, \ell_2, \ell_1' \in D$:*

$$\text{if } (s_1, \ell_1) \longrightarrow (s_2, \ell_2) \text{ and } \ell_1 \sqsubseteq \ell_1'$$
$$\text{then } \exists \ell_2' \in D : (s_1, \ell_1') \longrightarrow (s_2, \ell_2') \text{ with } \ell_2 \sqsubseteq \ell_2'$$

Note that the lattice $\mathcal{L}$ will often be a complete lattice as the meet $\sqcap$ operation is often useful for the transition relation. The monotonicity property arises naturally for many cases, e.g. if a Galois connection exists the soundness of the concretisation function γ will lead to a monotone transition relation.

Transitions are usually written as $(s, \ell) \longrightarrow (s', \ell')$ whenever $(s, \ell, s', \ell') \in \longrightarrow$. Configurations are pairs of the form (s, ℓ) where $s \in S$ and $\ell \in D$.

Definition 25 (Lattice Transition System)**.** *A lattice transition system over a lattice automaton $\mathcal{T}$ is given by $(S \times D, \longrightarrow, (s_0, \ell_0))$ where*

- *$S \times D$ is the set of configurations,*
- *$\longrightarrow$ is the transition relation,*
- *and (s_0, ℓ_0) is the initial state.*

Definition 26 (Path)**.** *A finite path over a lattice automaton $\mathcal{T}$ is a finite sequence $\sigma = (s_0, \ell_0)(s_1, \ell_1) \cdots (s_n, \ell_n)$ such that $(s_i, \ell_i) \longrightarrow (s_{i+1}, \ell_{i+1})$ for all i, $0 \leq i \leq n-1$.*

The $\sqsubseteq$ ordering is extended to configurations such that

$$(s, \ell) \sqsubseteq (t, \ell') \iff s = t \wedge \ell \sqsubseteq \ell'$$

Given a set of configurations X and a configuration (s, ℓ) the notation $(s, \ell) \sqsubseteq X$ will mean that $\exists (s, \ell') \in X : \ell \sqsubseteq \ell'$.

For use in computing counter-examples abstract paths will be useful.

Definition 27 (Abstract Paths)**.** *A finite abstract path over a lattice automaton $\mathcal{T}$ is a finite sequence $\sigma = (s_0, \ell_0)(s_1, \ell_1) \cdots (s_n, \ell_n)$ such that*

$$(s_i, \ell_i) \longrightarrow (s_{i+1}, \ell'_{i+1}) \text{ for some } \ell'_{i+1} \sqsubseteq \ell_{i+1}$$

for all i, $0 \leq i \leq n-1$.

In an abstract path some steps might be going to more abstract states than the transition relation itself allows; thus an abstract path might not be realisable in the transition system itself.

Lemma 5. *The symbolic statespace of a timed automata can be generated by a lattice automaton.*

Proof sketch. Given a timed automaton $\mathcal{A} = (L, V, \mathcal{C}, l_0, \rightarrow, I_{\mathcal{C}})$ the corresponding lattice automaton can be constructed as

- The locations of the lattice automaton will be the set $L \times (V \rightarrow Dom(V))$.

- The lattice will be the DBM abstract domain, $\mathcal{L}_{DBM}$ from Definition 19.
- The transition relation will be as given by the symbolic zone semantics with extrapolation, as in Lemma 3.

The requirement that the transition relation has the monotonicity property can be seen to be fulfilled by examining that each of the DBM operations in the successor computation are monotonic. □

The basic algorithm (Algorithm 4) on lattice automata is to compute a covering set, $P \subseteq S \times \mathcal{L}$ such that for any reachable state (s, ℓ) in the transition system of the lattice automaton it holds that $(s, \ell) \sqsubseteq (s, \ell')$ for some $(s, \ell') \in P$. The algorithm can optionally be given a logic formulae ϕ which is the state property the user wishes to prove; if ϕ is given and found to be false, a counterexample can be returned.

Algorithm 4 Algorithm to compute a covering set or a counter-example, given a model in the form of a lattice transition system $\mathcal{M} = (S, \mathcal{L}, \rightarrow)$, initial configuration (s_0, ℓ_0) and formulae ϕ, and using lattice join as abstraction.

1: **procedure** MC-JOIN($\mathcal{M}, (s_0, \ell_0), \phi$)
2: $W := \{(s_0, \ell_0)\}, P := \emptyset$
3: **while** $W \neq \emptyset$ **do**
4: Remove some (s, ℓ) from W
5: **if** $(s, \ell) \models \phi$ **then return** counterexample
6: **if** $(s, \ell) \not\sqsubseteq P$ **then**
7: **for all** (t, ℓ') s.t. $(s, \ell) \rightarrow (t, \ell')$ **do**
8: $\ell'' := \ell' \sqcup \bigsqcup\{\ell''' | (t, \ell''') \in W \cup P\}$
9: $W := W \setminus \{(t, \ell''') | \ell''' \sqsubseteq \ell''\} \cup \{(t, \ell'')\}$
10: $\ell'' := \ell \sqcup \bigsqcup\{\ell''' | (s, \ell''') \in P\}$
11: $P := P \setminus \{(s, \ell') | \ell' \sqsubseteq \ell''\} \cup \{(s, \ell'')\}$
12: **return** Covering set P

The counter-example returned is an abstract path given by backtracking from the offending $(s, \ell) \not\models \phi$ to the initial location s_0. The counter-example is the abstract path given by:

$$(s_0, P(s_0))(s_1, P(s_1)) \ldots (s, \ell)$$

If the covering set computed by Algorithm 4 is too coarse for the purpose at hand, or an abstract path is returned as a counter-example but the abstract path is spurious, a *joining strategy* can help by refining the covering set.

Definition 28 (Joining Strategy)**.** *A joining strategy is a function*

$$\delta : (S \times \mathcal{L}) \times (S \times \mathcal{L}) \to \{true, false\}$$

detailing whether two states in a lattice transition system are allowed to be joined, or should be kept separate.

The joining strategy can be integrated in the cover algorithm (Algorithm 4) as done in Algorithm 5.

Algorithm 5 Algorithm to compute a covering set or a counter-example, given a model in the form of a lattice transition system $\mathcal{M} = (S, \mathcal{L}, \to)$, initial configuration (s_0, ℓ_0) and formulae ϕ, and using the joining strategy δ as abstraction.

1: **procedure** MC-JOIN-STRATEGY($\mathcal{M}, (s_0, \ell_0), \phi, \delta$)
2: $W := \{(s_0, \ell_0)\}, P := \emptyset$
3: **while** $W \neq \emptyset$ **do**
4: Remove some (s, ℓ) from W
5: **if** $(s, \ell) \models \phi$ **then return** counterexample
6: **if** $(s, \ell) \not\sqsubseteq P$ **then**
7: **for all** (t, ℓ') s.t. $(s, \ell) \to (t, \ell')$ **do**
8: $\ell'' := \ell' \sqcup \bigsqcup\{\ell''' | (t, \ell''') \in W \cup P, \delta(t, \ell''', t, \ell')\}$
9: $W := W \setminus \{(t, \ell''') | \ell''' \sqsubseteq \ell''\} \cup \{(t, \ell'')\}$
10: $\ell'' := \ell \sqcup \bigsqcup\{\ell''' | (s, \ell''') \in P, \delta(s, \ell''', s, \ell)\}$
11: $P := P \setminus \{(s, \ell') | \ell' \sqsubseteq \ell''\} \cup \{(s, \ell'')\}$
12: **return** Covering set P

Although the joining strategy is defined as a static function, nothing prevents it from being dynamic, i.e. taking the runtime properties of the algorithm into account.

In Chapter 4 the `opaal` tool implementing Algorithm 4 and Algorithm 5 will be introduced, along with a number of case studies. The case of timed automata will be covered in Chapter 5. In Chapter 7 it will be shown how lattice automata can be derived from a program as an abstract interpretation. The covering set of such a lattice automaton will then be isomorphic to the fix-point as computed by the classic work-list algorithm, Algorithm 2.

Chapter 3

Thesis Summary

This thesis is based on five papers. In general the papers have only been adjusted to fit the layout of this thesis, except where otherwise noted. Each paper will now be summarised by its abstract, contribution, and details about its publication.

In addition to these five papers, the following papers have also been published during the PhD study:

[44] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis Using Model Checking. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 114–124, 2010.

[84] M. C. Olesen, R. R. Hansen, J. Lawall, and N. Palix. Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Coding Standard certification. In *Pre-proceedings of the 4th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert)*, volume 33 of *Electronic Communications of the EASST*, pages 51–69, 2010.

[29] J. Brauer, R. R. Hansen, S. Kowalewski, K. G. Larsen, and M. C. Olesen. Adaptable Value-Set Analysis for Low-Level Code. In *Proceedings of the 6th International Workshop on Systems Software Verification (SSV)*, pages 32–43, 2011.

[63] T. Jensen, H. Pedersen, M. C. Olesen, and R. R. Hansen. THAPS: Automated Vulnerability Scanning of PHP Applications. In *Proceedings of the 17th Nordic Conference on Secure IT Systems (NordSec)*, volume 7617 of Lecture Notes in Computer Science, pages 31–46. Springer, 2012.

[85] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix. Coccinelle: Tool support for automated CERT C Secure Coding Standard certification. *Science of Computer Programming (SCP)*, 2012.

[23] S. Biallas, M. C. Olesen, F. Cassez, and R. Huuck. PtrTracker: Pragmatic Pointer Analysis. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, (to appear). IEEE Computer Society, 2013.

Paper A – `opaal`: A Lattice Model Checker

Andreas Engelbredt Dalsgaard René Rydhof Hansen
Kenneth Yrke Jørgensen Kim Guldstrand Larsen
Mads Chr. Olesen Petur Olsen Jiří Srba

We present a new open source model checker, `opaal`, for automatic verification of models using lattice automata. Lattice automata allow the users to incorporate abstractions of a model into the model itself. This provides an efficient verification procedure, while giving the user fine-grained control of the level of abstraction by using a method similar to Counter-Example Guided Abstraction Refinement. The `opaal` engine supports a subset of the uppaal timed automata language extended with lattice features. We report on the status of the first public release of `opaal`, and demonstrate how `opaal` can be used for efficient verification on examples from domains such as database programs, lossy communication protocols and cache analysis.

Contributions

- Exploration of the use of the lattice automata formalism for verification, for a diverse set of domains.
- The implementation of the prototype model checker `opaal`.
- Experiments showing the potential benefits of using lattice automata.

Publication history

The paper was accepted and presented at the NASA Formal Methods - Third International Symposium (NFM 2011), and published in the Springer Lecture Notes in Computer Science vol. 6617, p. 487–493. The paper has been reformatted to fit the layout of this thesis.

Paper B – Multi-Core Reachability for Timed Automata

Andreas Engelbredt Dalsgaard Alfons Laarman
Kim G. Larsen Mads Chr. Olesen Jaco van de Pol

Model checking of timed automata is a widely used technique. But in order to take advantage of modern hardware, the algorithms need to be parallelized. We present a multi-core reachability algorithm for the more general class of well-structured transition systems, and an implementation for timed automata.

Our implementation extends the `opaal` tool to generate a timed automaton successor generator in C++, that is efficient enough to compete with the UPPAAL model checker, and can be used by the discrete model checker LTSmin, whose parallel reachability algorithms are now extended to handle subsumption of semi-symbolic states. The reuse of efficient lockless data structures guarantees high scalability and efficient memory use.

With experiments we show that `opaal`+LTSmin can outperform the current state-of-the-art, UPPAAL. The added parallelism is shown to reduce verification times from minutes to mere seconds with speedups of up to 40 on a 48-core machine. Finally, strict BFS and (surprisingly) parallel DFS search order are shown to reduce the state count, and improve speedups.

Contributions

- A multi-core model-checker exploiting subsumption, either as lattice automata or more generally for well-structured transition systems.
- Extension of the `opaal` tool to be a frontend for LTSmin.
- Implementing efficient data structures in LTSmin, for handling semi-symbolic states.
- Experiments showing the performance and scalability of the tool.

Publication history

The paper was accepted and presented at the 10th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2012), and published in the Springer Lecture Notes in Computer Science vol. 7595, p. 91–106. The paper has been reformatted to fit the layout of this thesis, and it has been clarified that the reachable part of the transition system needs to be finite.

Paper C – Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction

Alfons Laarman Mads Chr. Olesen
Andreas Engelbredt Dalsgaard Kim Guldstrand Larsen
Jaco van de Pol

This paper contributes to the multi-core model checking of timed automata (TA) with respect to liveness properties, by investigating checking of TA Büchi emptiness under the very coarse inclusion abstraction or zone subsumption, an open problem in this field.

We show that in general Büchi emptiness is not preserved under this abstraction, but some other structural properties are preserved. Based on those, we propose a variation of the classical nested depth-first search (NDFS) algorithm that exploits subsumption. In addition, we extend the multi-core CNDFS algorithm with subsumption, providing the first parallel LTL model checking algorithm for timed automata.

The algorithms are implemented in LTSmin, and experimental evaluations show the effectiveness and scalability of both contributions: subsumption halves the number of states in the real-world FDDI case study, and the multi-core algorithm yields speedups of up to 40 using 48 cores.

Contributions

- A multi-core implementation of the nested depth-first-search algorithm for timed Büchi automata.
- Proving that subsumption does not preserve Büchi emptiness.
- Proving that subsumption preserves some structural properties.
- An NDFS algorithm exploiting the properties preserved by subsumption.
- Experiments showing the efficiency and scalability of the algorithm.

Publication history

The paper was accepted at the 25th International Conference on Computer Aided Verification (CAV 2013), and published in the Springer Lecture Notes in Computer Science vol. 8044, p. 968–983. The paper has been reformatted to fit the layout of this thesis.

Paper D – An Automata-Based Approach to Trace Partitioned Abstract Interpretation

Mads Chr. Olesen René Rydhof Hansen
Kim Guldstrand Larsen

Trace partitioning is a technique for retaining precision in abstract interpretation, by partitioning all traces into a number of classes and computing an invariant for each class. In this work we present an automata-based approach to trace partitioning, by augmenting the finite automaton given by the control-flow graph with abstract transformers over a lattice. The result is a lattice automaton, for which efficient model-checking tools exist. By adding additional predicates to the automaton, different classes of traces can be distinguised.

This shows a very practical connection between abstract interpretation and model checking: a formalism encompassing problems from both domains, and accompanying machinery that can be used to solve problems from both domains efficiently.

This practical connection has the advantage that improvements from one domain can very easily be transferred to the other. We exemplify this with the use of multi-core processors for a scalable computation. Furthermore, the use of a modelling formalism as intermediary format allows the program analyst to simulate, combine and alter models to perform ad-hoc experiments.

Contributions

- Showing how an abstract interpretation can be viewed as an lattice automata.
- Showing that the covering set algorithm for lattice automata computes the same result as the fix-point computation of an abstract interpretation.
- Exploring how trace partitioning is naturally incorporated in the lattice automata model.
- A prototype to generate `opaal` lattice automata from C programs, using the octagon domain implementation from APRON, extension of the LTSmin multi-core backend with joining, and experiments showing that the use of LTSmin as a multi-core backend is scalable under trace partitioning.

Publication history

The paper is currently under submission.

Paper E – What is a Timing Anomaly?

Franck Cassez René Rydhof Hansen
Mads Chr. Olesen

Timing anomalies make worst-case execution time analysis much harder, because the analysis will have to consider all local choices. It has been widely recognised that certain hardware features are timing anomalous, while others are not. However, defining formally what a timing anomaly is, has been difficult.

We examine previous definitions of timing anomalies, and identify examples where they do not align with common observations. We then provide a definition for *consistently slower hardware traces* that can be used to define timing anomalies and aligns with common observations.

Contributions

- Exploring previous formal definitions of timing anomalies, comparing and contrasting them against one another, and intuitive definitions.
- Presenting different hardware models to highlight the differences of the previous definitions, exposing counter-intuitive examples.
- Showing that input data should not be deemed timing anomalous.
- Proposing a definition of timing anomalies not relying on abstraction, showing how this correlates better with the intuitive definition.

Publication history

The paper was accepted and presented at the 12th International Workshop on Worst-Case Execution-Time Analysis (WCET 2012), and published in the Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik OASICS vol. 23, p. 1–12. The paper has been reformatted to fit the layout of this thesis.

Chapter 4

opaal: A Lattice Model Checker

This chapter is based on the paper "opaal: A Lattice Model Checker" [42].

The paper explores model checking statespaces with lattice structure, and presents the opaal tool which is the basis for the subsequent contributions of this thesis. The opaal tool was originally built as a prototype model checker, compatible with UPPAAL, but extended to more general lattices than just DBMs. It allowed prototyping of various methods such as joining strategies, while still allowing the input format to be edited in the UPPAAL graphical editor.

Abstract

We present a new open source model checker, opaal, for automatic verification of models using lattice automata. Lattice automata allow the users to incorporate abstractions of a model into the model itself. This provides an efficient verification procedure, while giving the user fine-grained control of the level of abstraction by using a method similar to Counter-Example Guided Abstraction Refinement. The opaal engine supports a subset of the UPPAAL timed automata language extended with lattice features. We report on the status of the first public release of opaal, and demonstrate how opaal can be used for efficient verification on examples from domains such as database programs, lossy communication protocols and cache analysis.

4.1 Introduction

Common to almost all applications of model checking is the notion of an underlying concrete system with a very large—or sometimes even infinite—concrete state space. In order to enable model checking of such systems, it

is necessary to construct an abstract model of the concrete system, where some system features are only modelled approximately and system features that are irrelevant for a given verification purpose are "abstracted away".

The opaal model checker described in this paper allows for such abstractions to be integrated in the model through user-defined lattices. Models are formalised by *lattice automata*: synchronising extended finite state machines which may include lattices as variable types. The lattice elements are ordered by the amount of behaviour they induce on the system, that is, larger lattice elements introduce more behaviour. We call this the *monotonicity property*. The addition of explicit lattices makes it possible to apply some of the advanced concepts and expressive power of abstract interpretation directly in the models.

Lattice automata, as implemented in opaal, are a subclass of well-structured transition systems [51]. The tool can exploit the ordering relation to reduce the explored state space by not re-exploring a state if its behaviour is *covered* by an already explored state. In addition to the ordering relation, lattices have a *join operator* that joins two lattice elements by computing their least upper bound, thereby potentially overapproximating the behaviour, with the gain of a reduced state space. Model checking the overapproximated model can however be inconclusive. We introduce the notion of a *joining strategy* affording the user more control over the overapproximation, by specifying which lattice elements are joinable. This allows for a form of user-directed CEGAR (Counter-Example Guided Abstraction Refinement) [59, 10]. The CEGAR approach can easily be automated by the user, by exploiting application-specific knowledge to derive more fine-grained joining strategies given a spurious error trace. Thus providing, for some systems and properties, efficient model checking and conclusive answers at the same time.

The opaal model checker is released under an open source license, and can be freely downloaded from our webpage: www.opaal-modelchecker.com. The tool is available both in a GUI and CLI version, shown in Figure 4.1. The UPPAAL [16] GUI is used for creation of models.

The opaal tool is implemented in Python and is a stand-alone model checking engine. Models are specified using the UPPAAL XML format, extended with some specialised lattice features. Using an interpreted language has the advantage that it is easy to develop and integrate new lattice implementations in the core model checking algorithm. Our experiments indicate that although opaal uses an interpreted language, it is still sufficiently fast to be useful.

Users can create new lattices by implementing simple Python class interfaces. The new classes can then be used directly in the model (including all user-defined methods). Joining strategies are defined as Python functions.

An overview of the opaal architecture is given in Figure 4.2, showing the five main components of opaal. The "Successor Generator" is responsible

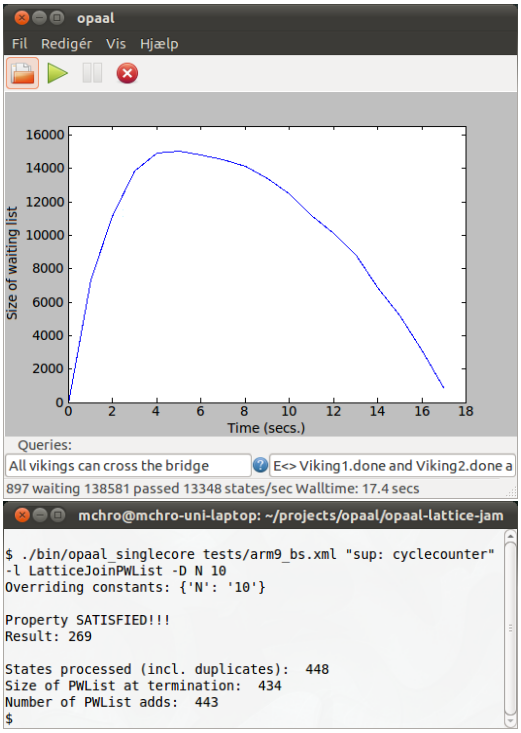


Figure 4.1: `opaal` GUI and CLI

for generating a transition function for the transition system based on the semantics of UPPAAL automata. The transition function is combined with one or more lattice implementations from the "Lattice Library".

The "Successor Generator" exposes an interface that the "Reachability Checker" can use to perform the actual verification. During this process a "Passed-Waiting List" is used to save explored and to-be explored states; it employs a user-provided "Joining Strategy" on the lattice elements of states, before they are added to the list.

4.2 Examples

In this section we present a few examples to demonstrate the wide applicability of `opaal`. The tool currently has a number of readily available lattices that are used to abstract the real data in our examples.

4.2.1 Database Programs

In recent work by Olsen et al. [86], the authors propose using present-absent sets for the verification of database programs. The key idea is that many behavioural properties may be verified by only keeping track of a few

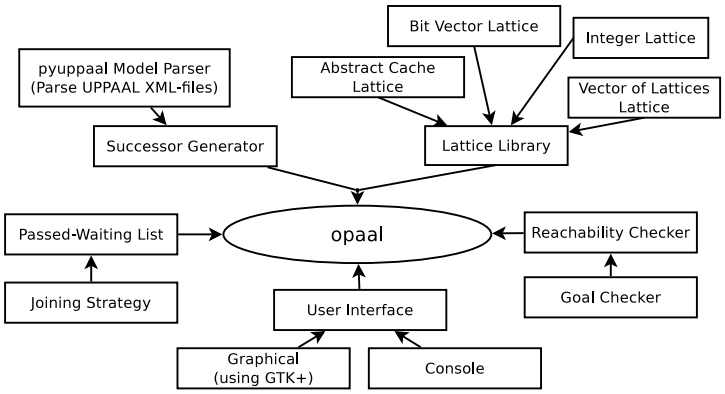


Figure 4.2: Overview of `opaal`'s architecture.

representative data values.

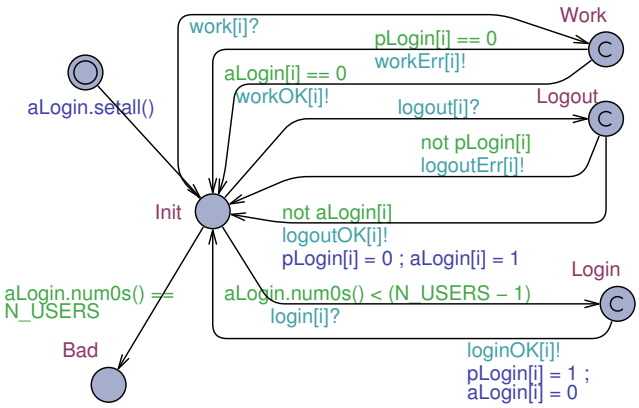
This idea can be naturally described as a lattice tracking the definite present- and absent-ness of database elements. In the model, this is implemented using a bit-vector lattice. For the experiment we adopt a model from [86], where users can login, work, and logout. The model has been updated to fit within the lattice framework, as shown in Figure 4.3(a). In the code in Figure 4.3(b), the construct **extern** is used on line 3 to import a lattice from the library. Subsequently two lattice variables, pLogin and aLogin, are defined at line 4 and 5, both vectors of size N_USERS. The lattice variables are used in the transitions of the graphical model, where e.g. a special method "num0s()" is used to count the number of 0's in the bitvector. The definition of a lattice type in Figure 4.3(c) is just an ordinary Python class with at least two methods: join and the ordering.

We can verify that two users of the system cannot work at the same time using explicit exploration, or by exploiting the lattice ordering to do cover checks, see Figure 4.4.

Another property to check is that the database cannot become full. For this property we can exploit a CEGAR approach: A naïve joining strategy will give inconclusive results, but refining the joining strategy not to join two states if the resulting state has a full database, leads to conclusive results while still preserving a significant speedup, see Figure 4.5.

4.2.2 Asynchronous Lossy Communication Protocol: Leader Election

Communication protocols where messages are asynchronously passed via an unreliable (lossy and duplicating) medium can be modelled as a lattice automaton. As long as we are interested in safety properties, such a communication can be modelled as a set of already sent messages called *pool*. Initially the set *pool* is empty. Once a message it sent, it is added to the set



(a) Database model

```
1 const int N_USERS = 17;
2 ...
3 extern IntersBitVector;
4 IntersBitVector pLogin[N_USERS];
5 IntersBitVector aLogin[N_USERS];
```

(b) Lattice variables

```
1 class IntersBitVector:
2     def join(self, other):
3         ...
4
5     def __le__(self, other):
6         ...
```

(c) Lattice library (in Python

Figure 4.3

pool and it remains there forever (duplication). As the protocol parties are not forced to read any message from *pool* and we ask about safety properties, lossiness is covered by the definition too.

It is obvious that 2^{pool}, i.e. the set of all subsets of *pool*, together with the subset ordering is a complete lattice. As long as the set of messages is finite and all parties in the protocol behave in the way that their steps are conditioned only on the presence of a message in the pool and not on its absence, the system will satisfy the monotonicity property and we can apply our model checker.

We have modelled the asynchronous leader election protocol [52] in `opaal`. Here we have N agents with their unique identifications $0, 1, \ldots, N-1$ and they select a leader with the highest id. Experimental data, for the property that only the agent with the highest id can become leader, are provided in Figure 4.6. The cover check column refers to using only the monotonicity property to reduce the explored state-space. We can see that while being exact (no overapproximation), the speed-up is considerable. Moreover, using the join strategy provides even more significant speed-up while still providing conclusive answers.

Number of users	explicit exploration	cover check
2	224 (<1s)	56 (<1s)
3	2352 (2s)	336 (<1s)
4	21952 (28s)	1792 (2s)
5	192080 (8:22m)	8960 (9s)
6	-	43008 (48s)
7	-	200704 (4:38m)

Figure 4.4: Explored states and time for the property "no two users work at the same time"

Number of users	explicit exploration	joining (naïve strategy)	joining (refined strategy)
8	6312 (15s)	(Inconclusive) 51 (<1s)	787 (1s)
9	14228 (56s)	(Inconclusive) 57 (<1s)	1238 (2s)
10	31614 (4:19m)	(Inconclusive) 63 (<1s)	976 (2s)
11	69478 (21:35m)	(Inconclusive) 69 (<1s)	1036 (2s)
12	-	(Inconclusive) 75 (<1s)	1707 (3s)
16	-	(Inconclusive) 99 (<1s)	25900 (4:18m)
17	-	(Inconclusive) 105 (<1s)	66490 (25:01m)

Figure 4.5: Explored states and time for the property "database cannot become full"

Number of agents	explicit exploration	cover check	joining
5	840 (5s)	37 (<1s)	17 (<1s)
6	5760 (5:20m)	58 (<1s)	23 (<1s)
7	45360 (671:02m)	86 (1s)	30 (<1s)
15	-	682 (4:21m)	122 (2s)
25	-	2927 (283:16m)	327 (12s)
50	-	-	1277 (4:19m)
100	-	-	5052 (98:45m)

Figure 4.6: Explored states and time for the leader election protocol

4.2.3 Cache Analysis

To ensure safe scheduling of real-time systems, the estimation of Worst-Case Execution Time (WCET) of each task in a given system is necessary [99]. One major part of determining WCETs for modern processors is accounting for the effects of the memory cache. Efficient abstractions exist for analysing

some types of caches [3], which we have implemented as a lattice. By recasting the cache analysis into our framework we gain the ability to give WCET guarantees, and gradually refine those guarantees by being more and more concrete with respect to the data-flow of the program.

On a simple program (binary search in array of size 100) and a simple cache we get the same WCET using all approaches. The complete state space has 5726 states (computed in 6s), cover update reduces this to 4043 states (3s), while join only needs to store 3944 states ($3s$). On more complex examples join will start to give overapproximated guarantees, which can be further refined.

4.2.4 Timed Automata

It is well-known that the theory of *zones* of timed automata (see e.g. [58, 20]) is a finite-state abstraction of clock values with a lattice structure. A zone-lattice is currently being developed for use in `opaal`, but has not matured to a point where meaningful experiments can be made yet.

4.3 Conclusion

We presented a new model checker, `opaal`, for lattice automata and provided a number of applications. The expressiveness of the formalism, derived from well-structured transition systems, promises broad applicability of the tool. Our initial experiments indicate that careful abstraction using the techniques implemented in `opaal` lead to efficient verification.

We plan on extending the foundations of `opaal` to additional formalisms such as Petri nets, as well as on improving the performance of the tool by rewriting core parts in a compiled language. Of course, additional lattices and areas of application are also to be investigated.

Chapter 5

Efficient Multi-Core Reachability Checking for Timed Automata

This chapter is based on the paper "Multi-Core Reachability for Timed Automata" [43].

As was shown in Chapter 2 solving the reachability problem for timed automata is typically done by an (implicit) abstract interpretation – as in the following. Therefore, any improvements to the method for timed automata (such as a multi-core model checking algorithm) should be transferable back into the more general setting of other abstract interpretations. One important aspect is that the use of joining in this paper is limited to the set-join of the disjunctive completion (Definition 23), a limitation that will be lifted in Chapter 7.

Additionally, the paper is written in the more general framework of Well-Structured Transition Systems, which has less structure than a lattice – the ordering is not a partial order (see Definition 8) but a well-quasi order (see Definition 29). Of course the work still holds for the more structure present under a partial order.

Abstract

Model checking of timed automata is a widely used technique. But in order to take advantage of modern hardware, the algorithms need to be parallelized. We present a multi-core reachability algorithm for the more general class of well-structured transition systems, and an implementation for timed automata.

Our implementation extends the `opaal` tool to generate a timed automaton successor generator in C++, that is efficient enough to compete with

the UPPAAL model checker, and can be used by the discrete model checker LTSMIN, whose parallel reachability algorithms are now extended to handle subsumption of semi-symbolic states. The reuse of efficient lockless data structures guarantees high scalability and efficient memory use.

With experiments we show that `opaal`+LTSMIN can outperform the current state-of-the-art, UPPAAL. The added parallelism is shown to reduce verification times from minutes to mere seconds with speedups of up to 40 on a 48-core machine. Finally, strict BFS and (surprisingly) parallel DFS search order are shown to reduce the state count, and improve speedups.

5.1 Introduction

In industries developing safety-critical real-time systems, a number of safety requirements must be fulfilled. Model checking is a well-known method to achieve this and is critical for ensuring correct behaviour along all paths of execution of a system. One popular formalism for real-time systems is timed automata [5], where the time is modelled as a number of resettable clocks. Good tool support for timed automata exists [16].

However, as the desire to model check ever larger and more complex models arises, there is a need for more effective techniques. One option for handling large models has always been to buy a bigger machine. This provided great improvements; while early model checkers handled thousands of states, now we can handle billions. However, in recent years processor speed has stopped increasing, and instead more cores are added. These cores cannot be taken advantage of by the normal sequential algorithms for model checking.

The goal of this work is to develop scaling multi-core reachability for timed automata [5] as a first step towards full multi-core LTL model checking. A review of the history of discrete model checkers shows that indeed multi-core reachability is a crucial ingredient for efficient parallel LTL model checking (see Section 5.2). To attain our goal, we extended and combined several existing software tools:

LTSmin is a language-independent model checking framework, comprising, inter alia, an explicit-state multi-core backend [68, 24].

`opaal` is a model checker designed for rapid prototype implementation of new model checking concepts. It supports a generalised form of timed automata [42], and uses the UPPAAL input format.

The UPPAAL DBM library is an efficient library for representing timed automata zones and operations thereon, used in the UPPAAL model checker [16].

Contributions: We describe a multi-core reachability algorithm for timed automata, which is generalizable to all models where a well-quasi-ordering on the behaviour of states exist [51]. The algorithm has been implemented for timed automata, and we report on the structure and performance of this prototype.

Before we move on to a description of our solution and its evaluation, we first review related work, and then briefly introduce the modelling formalism.

5.2 Related Work

One efficient model checker for timed automata is the UPPAAL tool [16, 13]. Our work is closely related to UPPAAL in that we share the same input format and reuse its editor to create input models. In addition, we reused the open source UPPAAL DBM library for the internal symbolic representation of time zones.

Distributed model checking algorithms for timed automata were introduced in [18, 12]. These algorithms exhibited almost linear scalability (50–90% efficiency) on a 14-node cluster of that time. However, analysis also shows that static partitioning used for distribution has some inherent limitations [28]. Furthermore, in the field of explicit-state model checking, the DiVinE tool showed that static partitioning can be reused in a shared-memory setting [11]. While the problem of parallelisation is considerably simpler in this setting, this tool nonetheless featured suboptimal performance with less than 40% efficiency on 16-core machines [69]. It was soon demonstrated that shared-memory systems are exploited better by combining local search stacks with a lockless hash table as shared passed set and an off-the-shelf load balancing algorithm for workload distribution [69]. Especially in recent experiments on newer 48-core machines [49, Sec. 5], the latter solution was clearly shown to have the edge with 50–90% efficiency.

Linear-time, on-the-fly liveness verification algorithms are based on depth-first search (DFS) order [71]. Next to the additional scalability, the shared hash table solution also provides more freedom for the search algorithm, which can be pseudo DFS and pseudo breadth-first search (BFS) order [69], but also strict BFS (see Subsection 5.6.2). This freedom has already been exploited by parallel NDFS algorithms for LTL model checking [71, 49] that are linear in the size of the input graph (unlike their BFS-based counterparts). While these algorithms are heuristic in nature, their scalability has been shown to be superior to their BFS-based counterparts.

5.3 Preliminaries

We will now define the general formalism of well-structured transition systems [51, 1], and specifically networks of timed automata under the zone

abstraction [4].

Definition 29 (Well-quasi-ordering)**.** *A well-quasi-ordering $\sqsubseteq$ is a reflexive and transitive relation over a set X, s.t. for any infinite sequence $x_0, x_1, \ldots$ eventually for some $i < j$ it will hold that $x_i \sqsubseteq x_j$.*

In other words, in any infinite sequence eventually an element exists which is "larger" than some earlier element.

Definition 30 (Well-structured transition system)**.** *A well-structured transition system is a 3-tuple $(S, \rightarrow, \sqsubseteq)$, where S is the set of states, $\rightarrow$: $S \times S$ is the (computable) transition relation and $\sqsubseteq$ is a well-quasi-ordering over S, s.t. if $s \rightarrow t$ then $\forall s'.s \sqsubseteq s'$ there $\exists t'.s' \rightarrow t' \wedge t \sqsubseteq t'$.*[1]

We thus require $\sqsubseteq$ to be a monotonic ordering on the behaviour of states, i.e., if $s \sqsubseteq t$ then t has at least the behaviour of s (and possibly more), and we say that t *subsumes* or *covers* s.

One instance of well-structured transition systems arise from the symbolic semantics of timed automata. Timed automata are finite state machines with a finite set of real-valued, resettable clocks. Transitions between states can be guarded by constraints on clocks, denoted $G(C)$.

Definition 31 (Timed automaton)**.** *A timed automaton is a 6-tuple $\mathcal{A} = (L, C, Act, s_0, \rightarrow, I_C)$ where*

- *L is a finite set of locations, typically denoted by ℓ*
- *C is a finite set of clocks, typically denoted by c*
- *Act is a finite set of actions*
- *$s_0 \in L$ is the initial location*
- *$\rightarrow \subseteq L \times G(C) \times Act \times 2^C \times L$ is the (non-deterministic) transition relation. We normally write $\ell \xrightarrow{g,a,r} \ell'$ for a transition, where ℓ is the source location, g is the guard over the clocks, a is the action, and r is the set of clocks reset.*
- *$I_C : L \rightarrow G(C)$ is a function mapping locations to downwards closed clock invariants.*

Using the definition of timed automata we can now define networks of timed automata, as modelled by UPPAAL, see [16] for details. A network of timed automata is a parallel composition of timed automata that enables synchronisation over a finite set of channel names $Chan$. We let $ch!$ and $ch?$ denote the output and input action on a channel $ch \in Chan$.

[1] With strong compatibility, see [51]

Definition 32 (Network of timed automata)**.** *Let* $Act = \{ch!, ch?|ch \in Chan\} \cup \{\tau\}$ *be a finite set of actions, and let* C *be a finite set of clocks. Then the parallel composition of timed automata*

$$\mathcal{A}_i = (L_i, C, Act, s_0^i, \rightarrow_i, I_C^i)$$

for all $1 \leq i \leq n$*, where* $n \in \mathbb{N}$*, is a network of timed automata, denoted* $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2||\ldots||\mathcal{A}_n$*.*

The concrete semantics of timed automata [16] gives rise to a possibly uncountable state space. To model check it a finite abstraction of the state space is needed; the abstraction used by most model checkers is the zone abstraction [27]. Zones are sets of clock constraints that can be efficiently represented by Difference Bounded Matrices (DBMs) [19]. The fundamental operations of DBMs are:

- $D \uparrow$ modifying the constraints such that the DBM represents all the clock valuations that can result from delay from the current constraint set
- $D \cap D'$ adding additional constraints to the DBM, e.g. because a transition is taken that imposes a clock constraint (guard clock constraints can also be represented as a DBM, and we will do so) [2]. The additional constraints might also make the DBM empty, meaning that no clock valuations can satisfy the constraints.
- $D[r]$ where $r \subseteq C$ is a clock reset of the clocks in r.
- D/B doing maximal bounds extrapolation, where $B : C \rightarrow \mathbb{N}_0$ is the maximal bounds needed to be tracked for each clock. Extrapolation with respect to maximal bounds [14] is needed to make the number of DBMs finite. Basically, it is a mapping for each clock indicating the maximal possible constant the clock can be compared to in the future. It is used in such a way that if the value of a clock has passed its maximal constant, the clock's value is indistinguishable for the model.
- $D \subseteq D'$ for checking if the constraints of D' imply the constraints of D, i.e. D' is a more relaxed DBM. D' has the behaviour of D and possibly more.

Lemma 6. *Timed automata under the zone abstraction are well-structured transition systems:* $(S, \Rightarrow_{DBM}, Act, \sqsubseteq)$ *s.t.*

1. S *consists of pairs* (ℓ, D) *where* $\ell \in L$*, and* D *is a DBM.*

[2] The DBM might need to be put into normal form after more constraints have been added [27]

2. $\Rightarrow_{DBM}$ *is the symbolic transition function using DBMs, and Act is as before*

3. $\sqsubseteq: S \to S$ *is defined as* $(\ell, D) \sqsubseteq (\ell', D')$ *iff* $\ell = \ell'$*, and* $D \subseteq D'$*.*

Remark that part of the ordering $\sqsubseteq$ is compared using discrete equality (the location vector), while only a subpart is compared using a well-quasi-ordering. Without loss of generality, and as done in [42], we can split the state into an explicit part $\mathcal{S}$, and a symbolic part Σ, s.t. the well-structured transition system is defined over $\mathcal{S} \times \Sigma$. We denote the explicit part as $s, t, r \in \mathcal{S}$ and the symbolic part of states by $\sigma, \tau, \rho, \pi, \upsilon \in \Sigma$, and a state as a pair (s, σ).

Model checking of safety properties is done by proving or disproving the reachability of a certain concrete goal location s_g.

Definition 33 ((Safety) Model checking of a well-structured transition system)**.** *Given a well-structured transition system* $(\mathcal{S} \times \Sigma, \to, \sqsubseteq)$*, an initial state* $(s_0, \sigma_0) \in \mathcal{S} \times \Sigma$*, and a goal location* s_g *does a path exist* $(s_0, \sigma_0) \to \cdots \to (s_g, \sigma'_g)$*.*

In practice, the transition system is constructed *on-the-fly* starting from (s_0, σ_0) and recursively applying $\to$ to discover new states. To facilitate this, we extend the next-state interface of PINS with subsumption:

Definition 34. *A next-state interface with subsumption has three functions:*
INITIAL-STATE$() = (s_0, \sigma_0)$*,*
NEXT-STATE$((s, \sigma)) = \{(s_1, \sigma_1), \ldots, (s_n, \sigma_n)\}$ *returning all successors of* (s, σ)*,* $(s, \sigma) \to (s_i, \sigma_i)$*, and*
COVERS$(\sigma', \sigma) = \sigma \sqsubseteq \sigma'$ *returning whether the symbolic part* σ' *subsumes* σ*.*

5.4 A Multi-Core Timed Reachability Tool

For the construction of our real-time multi-core model checker, we made an effort to reuse and combine existing components, while extending their functionality where necessary. For the specification models, we use the UPPAAL XML format. This enables the use of its extensive real-time modelling language through an excellent user interface. To implement the model's semantics (in the form of a next-state interface) we rely on `opaal` and the UPPAAL DBM library.[3] Finally, LTSMIN is used as a model checking backend, because of its language-independent design.

Figure 5.1 gives an overview of the new toolchain. It shows how the XML input file is read by `opaal` which generates C++ code. The C++ file implements the PINS interface with subsumption specifically for the input

[3] `http://people.cs.aau.dk/~adavid/UDBM/`

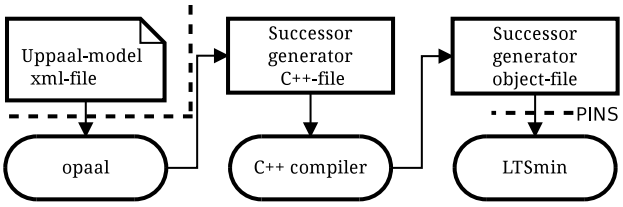


Figure 5.1: Reachability with subsumption [42]

model. Hence, after compilation (C++ compiler), LTSMIN can load the object file to perform the model checking.

Previously, the `opaal` tool was used to generate Python code [42], but important parts of its infrastructure, e.g., analysing the model to find max clock constants [14], can be reused. In Section 5.5, we describe how `opaal` implements the semantics of timed automata, and the structure of the generated C++ code.

The PINS interface of the LTSMIN tool [24] has been shown to enable efficient, yet language-independent, model checking algorithms of different flavours, inter alia: distributed [24], symbolic [24] and multi-core reachability [69, 70], and LTL model checking [71, 67]. We extended the PINS interface to distinguish the new symbolic states of the `opaal` successor generator according to Definition 34. In Section 5.6, we describe our new multi-core reachability algorithms with subsumption.

5.5 Successor Generation using opaal

The `opaal` tool was designed to rapidly prototype new model checking features and as such was designed to be extended with other successor generators. It already implements a substantial part of the UPPAAL features. For an explanation of the UPPAAL features see [16, p. 4-7]. The new C++ `opaal` successor generator supports the following features: templates, constants, bounded integer variables, arrays, selects, guards, updates, invariants on both variables and clocks, committed and urgent locations, binary synchronisation, broadcast channels, urgent synchronisation, selects, and much of the C-like language that UPPAAL uses to express guards and variable updates.

A state in the symbolic transition system using DBMs, is a location vector and a DBM. To represent a state in the C++ code we use a struct with a number of components: one integer for each location, and a pointer to a DBM object from the UPPAAL DBM library. Therefore a state is a tuple: $(\ell_1, \ldots, \ell_n, D)$.

The INITIAL-STATE function is rather straightforward: it returns a state struct initialised to the initial location vector, and a DBM representing

the initial zone (delayed, and with invariants applied as necessary). The structure of the NEXT-STATE function is more involved, because it needs to consider the syntactic structure of the model, as can be seen in Algorithm 6.

Algorithm 6 Overall structure of the successor generator

```
1  proc NEXT-STATE(s_in = (ℓ_1, ..., ℓ_n, D))
2  out_states := ∅
3  for ℓ_i ∈ ℓ_1, ..., ℓ_n
4    for all ℓ_i --g,a,r--> ℓ'_i
5      D' := D ∩ g
6      if D' ≠ ∅                                          ▷ is the guard satisfied?
7        if a = τ                                         ▷ this is not a synchronising transition
8          D' := D'[r]↑                                   ▷ clock reset, delay
9          D' := D' ∩ I^i_C(ℓ'_i) ∩ ⋂_{k≠i} I^k_C(ℓ_k)    ▷ apply clock invariants
10         if D' ≠ ∅
11           D' := D'/B(ℓ_1, ..., ℓ'_i ..., ℓ_n)
12           out_states := out_states ∪ {(ℓ_1, ..., ℓ'_i, ..., ℓ_n, D')}
13       else if a = ch!                                  ▷ binary sync. sender
14         for ℓ_j ∈ ℓ_1, ..., ℓ_n, j ≠ i
15           for all ℓ_j --g_j,ch?,r_j--> ℓ'_j            ▷ find receivers
16             if D'' := D' ∩ g_j ≠ ∅                     ▷ receiver guard satisfied?
17               D'' := D''[r][r_j]↑                      ▷ clock resets, delay
18               D'' := D'' ∩ I^i_C(ℓ'_i) ∩ I^j_C(ℓ'_j) ∩ ⋂_{k∉{i,j}} I^k_C(ℓ_k)   ▷ clock invariants
19               if D'' ≠ ∅
20                 D'' := D''/B(ℓ_1, ..., ℓ'_i, ..., ℓ'_j ..., ℓ_n)
21                 out_states := out_states ∪ {(l_1, ..., l'_i, ..., l'_j, ..., l_n, D'')}
22 return out_states
```

At l. 4, we consider all outgoing transitions for the current location of each process (l. 3). If the transition is internal, we can evaluate it right away, and possibly generate a successor at l. 12. If it is a sending synchronisation (`ch!`), we need to find possible synchronisation partners (l. 15). So again we iterate over all processes and the transitions of their current locations (l. 14–21).

In the generated C++ code a few optimisations have been made, compared to Algorithm 6: The loops on line l. 3 and l. 14 have been unrolled, since the number of processes they iterate over is known beforehand. In that manner the transitions to consider can be efficiently found. As an optimisation, before starting the code generation, we compute the set of all possible receivers for all channels, for the unrolling of l. 14. In practice there are usually many receivers but few senders for each channel, resulting in the unrolling being an acceptable trade-off.

When doing the max bounds extrapolation ($/$) in Algorithm 6, we obtain the bounds from a location-dependent function $B : L_1 \times \cdots \times L_n \to (C \to$

Algorithm 7 Reachability with subsumption [42]

```
1   proc reachability(s_g)
2     W := { INITIAL-STATE() }; P := ∅
3     while W ≠ ∅
4       W := W \ (s, σ) for some (s, σ) ∈ W
5       P := P ∪ {(s, σ)}
6       for (t, τ) ∈ NEXT-STATE((s, σ)) do
7         if t = s_g then report & exit
8         if ∄ρ: (t, ρ) ∈ W ∪ P ∧ COVERS(ρ, τ)
9           W := W \ {(t, ρ) | COVERS(τ, ρ)} ∪ (t, τ)
```

$\mathbb{N}_0$). This function is pre-computed in `opaal` using the method described in [14].

Some features are not formalised in this work, but have been implemented for ease of modelling. We support integer variables, urgency that can be modelled using urgent/committed locations and urgent channels, but also channel arrays with dynamically computed senders, broadcast channels, and process priorities. These are all implemented as simple extensions of Algorithm 6. Other features are supported in the form of a syntactic expansion, namely: selects, and templates.

To make the NEXT-STATE function thread-safe, we had to make the UPPAAL DBM library thread-safe. Therefore, we replaced its internal allocator with a concurrent memory allocator (see Section 5.7). We also replaced the internal hash table, used to filter duplicate DBM allocations, with a concurrent hash table.

5.6 Well-Structured Transition Systems in LTSmin

The current section presents the parallel reachability algorithm that was implemented in LTSMIN to handle well-structured transition systems with finite reachability sets. According to Definition 34, we can split up states into a discrete part, which is always compared using equality (for timed automata this consists of the locations and variables), and a part that is compared using a well-quasi-ordering (for timed automata this is the DBM).

We recall the sequential algorithm from [42] (Figure 7) and adapt it to use the next-state interface with subsumption. At its basis, this algorithm is a search with a waiting set (W), containing the states to be explored, and a passed set (P), containing the states that are already explored.

New successors (t, τ) are added to W (l. 9), but only if they are not subsumed by previous states (l. 8). Additionally, states in the waiting set W

that are subsumed by the new state are discarded (l. 9), avoiding redundant explorations.

5.6.1 A Parallel Reachability Algorithm with Subsumption

In the parallel setting, we localize all work sets (Q_p, for each worker p) and create a shared data structure L storing both W and P. We attach a status flag passed or waiting to each state in L to create a global view of the passed and waiting set and avoid unnecessary reexplorations. L can be represented as a multimap, saving multiple symbolic state parts with each explicit state part $L : \mathcal{S} \to \Sigma^*$. To make L thread-safe, we protect its operations with a fine-grained locking mechanism that locks only the part of the map associated with an explicit state part s: **lock**$(L(s))$, similar to the spinlocks in [69]. An off-the-shelf load balancer takes care of distributing work at the startup and when some Q_p runs empty prematurely. This design corresponds to the shared hash table approach discussed in Section 5.2 and avoids a *static partitioning* of the state space.

Algorithm 8 on page 68 presents the discussed design. The algorithm is initialised by calling reachability with the desired number of threads P and a discrete goal location s_g. This method initialises the shared data structure L and gets the initial state using the INITIAL-STATE function from the next-state interface with subsumption. The initial state is then added to L and the worker threads are initialised at l. 6. Worker thread 1 explores the initial state; work load is propagated later.

The **while** loop on l. 20 corresponds closely to the sequential algorithm, in a quick overview: a state (s, σ) is taken from the work set at l. 21, its flag is set to passed by grab if it were not already, and then the successors (t, τ) of (s, σ) are checked against the passed and the waiting set by update. We now discuss the operations on L (update, grab) and the load balancing in more detail.

To implement the subsumption check (line l. 8–9 in Figure 7) for successors (t, τ) and to update the waiting set concurrently, update is called. It first locks L on t. Now, for all symbolic parts and status flag ρ, f associated with t, the method checks if τ is already covered by ρ. In that case (t, τ) will not be explored. Alternatively, all ρ with status flag waiting that are covered by τ are removed from $L(t)$ and τ is added. The update algorithm maintains the invariant that a state in the waiting set is never subsumed by any other state in L: $\forall s\, \forall (\rho, f), (\rho', f') \in L(s) \colon f = \mathsf{waiting} \wedge \rho \neq \rho' \Rightarrow \rho \not\sqsubseteq \rho'$ (**Inv. 1**). Hence, similar to Figure 7 l. 8–9, it can never happen that (t, τ) first discards some (t, ρ) from $L(s)$ (l. 14) and is discarded itself in turn by some (t, ρ') in $L(s)$ (l. 10), since then we would have $\rho \sqsubseteq \tau \sqsubseteq \rho'$; by transitivity of $\sqsubseteq$ and the invariant, ρ and ρ' cannot be both in $L(t)$. Finally, notice that update unlocks $L(t)$ on all paths.

The task of the method grab is to check if a state (s, σ) still needs to be

Algorithm 9 Strict parallel BFS

```
1   proc search(s_0, σ_0, p)
2     C_p := if p = 1 then {(s_0, σ_0)} else ∅
3     do
4       while C_p ≠ ∅ ∨ balance(C_p)
5         C_p := C_p \ (s, σ) for some (s, σ) ∈ C_p
6           ...
7             N_p := N_p ∪ (t, τ)
8       load := reduce(sum, |N_p|, P)
9       C_p, N_p := N_p, ∅
10    while load ≠ 0
```

explored, as it might have been explored by another thread in the meantime. It first locks $L(s)$. If σ is no longer in $L(s)$ or it is no longer globally flagged waiting (l. 29), it is discarded (l. 22). Otherwise, it is "grabbed" by setting its status flag to passed. Notice again that on all paths through grab, $L(s)$ is unlocked.

Finally, the method balance handles termination detection and load balancing. It has the side-effect of adding work to Q_p. We use a standard solution [91].

5.6.2 Exploration Order

The shared hash table approach gives us the freedom to allow for a DFS or BFS exploration order depending on the implementation of Q_p. Note, however, that only pseudo-DFS/BFS is obtained, due to randomness introduced by parallelism.

It has been shown for timed automata that the number of generated states is quite sensitive to the exploration order and that in most cases strict BFS shows the best results [18]. Fortunately, we can obtain strict BFS by synchronising workers between the different BFS levels. To this end, we first split Q_p into two separate sets that hold the current BFS level (C_p) and the next BFS level (N_p) [2]. The order within these sets does not matter, as long as the current is explored before the next set. Load balancing will only be performed on C_p, hence a level terminates once $C_p = \emptyset$ for all p. At this point, if $N_p = \emptyset$ for all p, the algorithm can terminate because the next BFS level is empty. The synchronising reduce method counts $\sum_{i=1}^{P} |N_i|$ (similar to `mpi_reduce`).

Algorithm 9 shows a parallel strict-BFS implementation. An extra outer loop iterates over the levels, while the inner loop (l. 4–7) is the same as in Algorithm 8. Except for the lines that add and remove states to and from

the work set, which now operate on N_p and C_p. The (pointers to) the work sets are swapped, after the reduce call at l. 8 calculates the load of the next level.

5.6.3 A Data Structure for Semi-Symbolic States

In [69], we introduced a lockless hash table, which we reuse here to design a data structure for L that supports the operations used in Algorithm 8. To allow for massive parallelism on modern multi-core machines with steep memory hierarchies, it is crucial to keep a low memory footprint [69, Sec. II]. To this end, lookups in the large table of state data are filtered through a separate smaller table of hashes. The table assigns a unique number (the hash location) to each explicit state stored in it: $D\colon \mathcal{S} \to \mathbb{N}$. In finite reality, we have: $D\colon \mathcal{S} \to \{1, \ldots, N\}$.

We now reuse the state numbering of D to create a multimap structure for L. The first component of the new data structure is an array $I[N]$ used for indexing on the explicit state parts. To associate a set of symbolic states (pointers to DBMs) with our explicit state stored in $D[x]$, we are going to attach a linked list structure to $I[x]$. Creating a standard linked list would cause a single *cache line* access per element, increasing the memory footprint, and would introduce costly synchronisations for each modification. Therefore, we allocate multi-buckets, i.e., an array of pointers as one linked list element. To save memory, we store lists of just one element directly in I and completely fill the last multi-bucket.

Figure 5.2 shows three instances of the discussed data structure: L, L' and L''. Each multimap is a pointer (arrow) to an array I shown as a vertical bucket array. L contains $\{(s,\sigma),(t,\tau),(t,\rho),(t,\upsilon)\}$. We see how a multi-bucket with (fixed) length 3 is created for t, while the single symbolic state attached to s is kept directly in I. The figure shows how σ is moved when (s,π) is added by the add operation (dashed arrow), yielding L'. Adding π to t would have moved υ to a new linked multi-bucket together with π.

Removing elements from the waiting list is implemented by marking bucket entries as tombstone, so they can later be reused (see L''). This avoids memory fragmentation and expensive communication to reuse multi-

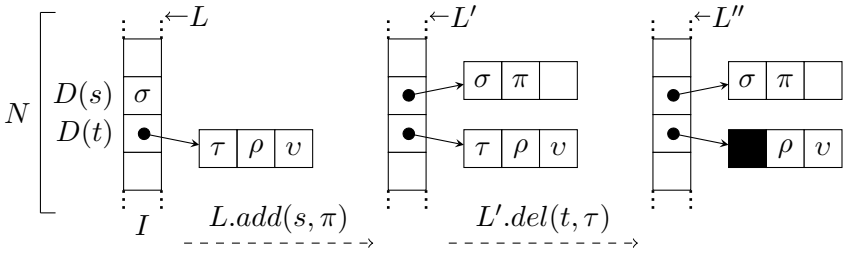


Figure 5.2: Data structure for L, and operations

```
struct link_or_dbm {
  bit pointer[60]
  bit flag ∈ {waiting, passed}
  bit lock ∈ {locked, unlocked}
  bit status[2] ∈ {empty, tomb,
                   dbm_ptr, list_ptr}
}
```

Figure 5.3: Bit layout of word-sized bucket

buckets. For highest scalability, we allocate multi-buckets of size 8, equal to a cache line. Other values can reduce memory usage, but we found this sufficiently efficient (see Section 5.7).

We still need to deal with locking of explicit states, and storing of the various flags for symbolic states (waiting/passed). Internally, the algorithms also need to distinguish between the different buckets: empty, tomb stone, linked list pointers and symbolic state pointers. To this end, we can bitcram additional bits into the pointers in the buckets, as is shown in Figure 5.3. Now **lock**$(L(s))$ can be implemented as a spinlock using the atomic *compare-and-swap* (CAS) instruction on $I[s]$ [69]. Since all operations on $L(s)$ are done after **lock**$(L(s))$, the corresponding bits of the buckets can be updated and read with normal load and store instructions.

5.6.4 Improving Scalability through a Non-Blocking Implementation

The size of the critical regions in Algorithm 8 depends crucially on the $|\Sigma|/|\mathcal{S}|$ ratio; a higher ratio means that more states in $L(t)$ have to be considered in the method update(t, τ), affecting scalability negatively. A similar limitation is reported for distributed reachability [28]. Therefore, we implemented a *non-blocking* version: instead of first deleting all subsumed symbolic states with a waiting flag, we atomically replace them with the larger state using CAS. For a failed CAS, we retry the subsumption check after a reread. L can be atomically extended using the well-known *read-copy-update* technique. However, workers might miss updates by others, as Inv. 1 no longer holds. This could cause $|\Sigma|$ to increase again.

5.7 Experiments

To investigate the performance of the generated code, we compare full reachability in `opaal`+LTSmin with the current state-of-the-art (UPPAAL).[4] To

[4] `opaal` is available at `https://code.launchpad.net/~opaal-developers/opaal/opaal-ltsmin-succgen`, LTSmin at `http://fmt.cs.utwente.nl/tools/ltsmin/`

investigate scalability, we benchmarked on a 48-core machine (a four-way AMD Opteron™ 6168) with a varying number of threads. Statistics on memory usage were gathered and compared against UPPAAL. Experiments were repeated 5 times.

We consider three models from the UPPAAL demos: `viking` (one discrete variable, but many synchronisations), `train-gate` (relatively large amount of code, several variables), and `fischer` (very small discrete part). Additionally, we experiment with a generated model, `train-crossing`, which has a different structure from most hand-made models. For some models, we created multiple numbered instances, the numbers represent the number of processes in the model.

For UPPAAL, we ran the experiments with BFS and disabled space optimisation. The `opaal_ltsmin` script in `opaal` was used to generate and compile models. In LTSMIN we used a fixed hash table (`--state=table`) size of 2^{26} states (`-s26`), waiting set updates as in Algorithm 8 (`-u1`) and multi-buckets of size 8 (`-l8`).

Performance & Scalability. Table 5.1 on page 69 shows the reachability runtimes of the different models in UPPAAL and `opaal`+LTSMIN with strict BFS (`--strategy=sbfs`). Except for `fischer6`, we see that both tools compete with each other on the sequential runtimes, with 2 threads however `opaal`+LTSMIN is faster than UPPAAL. With the massive parallelism of 48 cores, we see how verification tasks of minutes are reduced to mere seconds. The outlier, `fischer6`, is likely due to the use of more efficient clock extrapolations in UPPAAL, and other optimisations, as witnessed by the evolution of the runtime of this model in [17, 6].

We noticed that the 48-core runtimes of the smaller models were dominated by the small BFS levels at the beginning and the end of the exploration due to synchronisation in the load balancer and the `reduce` function. This overhead takes consistently 0.5–1 second, while it handles less than thousand states. Hence to obtain useful scalability measurements for small models, we excluded this time in the speedup calculations (Figure 5.4–5.7). The runtimes in Table 5.1–5.2 still include this overhead. Figure 5.4 plots the speedups of strict BFS with the standard deviation drawn as vertical lines (mostly negligible, hence invisible). Most models show almost linear scalability with a speedup of up to 40, e.g. `train-gate-N10`. As expected, we see that a high $|\Sigma|/|\mathcal{S}|$ ratio causes low scalability (see `fischer` and `train-crossing` and Table 5.1). Therefore, we tried the non-blocking variant (Subsection 5.6.3) of our algorithm (`-n`). As expected, the speedups in Figure 5.5 improve and the runtimes even show a threefold improvement for `fischer.6` (Table 5.2). The efficiency on 48 cores remains closely dependent to the $|\Sigma|/|\mathcal{S}|$ ratio of the model (or the average length of the lists in the multimap), but the scalability is now at least sub-linear and not stagnant anymore.

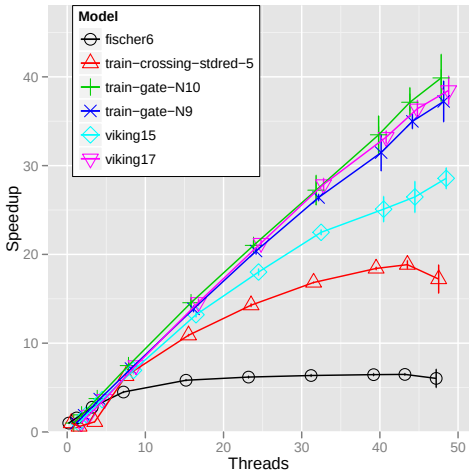


Figure 5.4: Speedup strict BFS

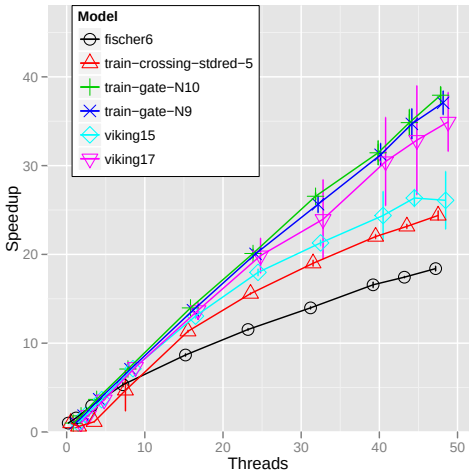


Figure 5.5: Speedup non-blocking strict BFS

We further investigated different search orders. Figure 5.6 shows results with pseudo BFS order (`--strategy=bfs`). While speedups become higher due to the lacking level synchronisations, the loose search order tends to reach "large" states later and therefore generates more states for two of the models ($|\Sigma_1|$ vs $|\Sigma_{48}|$ in Table 5.2). This demonstrates that our strict BFS implementation indeed pays off.

Finally, we also experimented with randomized DFS search order (`-prr --strategy=dfs`). Table 5.2 shows that DFS causes again more states to be generated. But, surprisingly, the number of states actually reduces with the parallelism for the `fischer6` model, even below the state count of strict BFS from Table 5.1! This causes a super-linear speedup in Figure 5.7 and

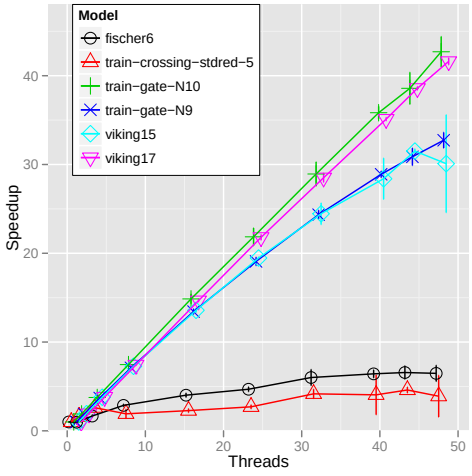


Figure 5.6: Speedup pseudo BFS

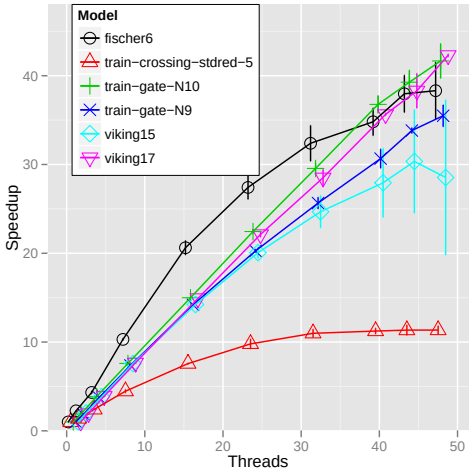


Figure 5.7: Speedup randomized pseudo DFS

threefold runtime improvement over strict BFS. We do not consider this behaviour as an exception (even though `train-crossing` does not show it), since it is compatible with our observation that parallel DFS finds shorter counter examples than parallel BFS [49, Sec. 4.3].

Design decisions. Some design decisions presented here were motivated by earlier work that has proven successful for multi-core model checking [69, 49]. In particular, we reused the shared hash table and a *synchronous* load balancer [91]. Even though we observed load distributions close to ideal, a modern work stealing solution might still improve our results, since the work granularity for timed reachability is higher than for untimed reachability. The main bottlenecks, however, have proven to be the increase in state count by parallelism and the cost of the spinlocks due to a high $|\Sigma|/|\mathcal{S}|$ ratio. The latter we partly solved with a non-blocking algorithm. Strict BFS orders have proven to aid the former problem and randomized DFS orders could aid both problems.

Memory usage. Table 5.3 on page 71 shows the memory consumption of UPPAAL (U-S0) and sequential `opaal`+LTSMIN (O+L_1) with strict BFS. From it, we conclude that our memory usage is within 25% of UPPAAL's for the larger models (where these measurements are precise enough). Furthermore, we extensively experimented with different concurrent allocators and found that TBB malloc (used in this paper) yields the best performance for our algorithms.[5] Its overhead (O+L_1 vs O+L_{48} in Table 5.3) appears to be limited to a moderate fixed amount of 250MB more than the sequential runs, for which we used the normal `glibc` allocator.

We also counted the memory usage inside the different data structures: the multimap L (including partly-filled multi-buckets), the hash table D, the combined local work sets (Q), and the DBM duplicate table (dbm). As we expected the overhead of the 8-sized multi-buckets is little compared to the size of D and the DBMs. We may however replace D with the compressed, parallel tree table (T) from [70]. The resulting total memory usage (O+L^T), can now be dominated by L, i.e., for `viking17`. But if we reduce L to a linked list (`-l2`), its size shrinks by 60% to 214MB for this model (L2). Just a modest gain compared to the total.

For completeness, we included the results of UPPAAL's state space optimisation (U-S2). As expected, it also yields great reductions, which is the more interesting since the two techniques are orthogonal and could be combined.

[5] cf. `http://fmt.cs.utwente.nl/tools/ltsmin/formats-2012/` for additional data

5.8 Conclusions

We presented novel algorithms and data structures for multi-core reachability on well-structured transition systems and an efficient implementation for timed automata in particular. Experiments show good speedups, up to 40 times on a 48-core machine and also identify current bottlenecks. In particular, we see speedups of 58 times compared to UPPAAL. Memory usage is limited to an acceptable maximum of 25% more than UPPAAL.

Our experiments demonstrate the flexibility of the search order that our parallel approach allows for. BFS-like order is shown to be occasionally slightly faster than strict BFS but is substantially slower on other models, as previously observed in the distributed setting. A new surprising result is that parallel randomized (pseudo) DFS order sometimes reduces the state count below that of strict BFS, yielding a substantial speedup in those cases.

Previous work has shown that better parallel reachability [69, 70] crucially enables new and better solutions to parallel model checking of liveness properties [71, 49]. Therefore, our natural next step is to port multi-core nested depth-first search solutions to the timed automata setting.

Because of our use of generic toolsets, more possibilities are open to be explored. The `opaal` support for the UPPAAL language can be extended and support for optimisations like symmetry reduction and partial order reduction could be added, enabling easier modeling and better scalability. Additionally, lattice-based languages [42] can be included in the C++ code generator. On the backend side, the distributed [24] and symbolic [24] algorithms in LTSMIN can be extended to support subsumption, enabling other powerful means of verification. We also plan to add a join operator to the PINS interface, to enable abstraction/refinement-based approaches [42].

Algorithm 8 Reachability with cover update of the waiting set

```
1  global L : S → (Σ × {waiting, passed})*
2  proc reachability(P, s_g)
3    L := S → ∅
4    (s_0, σ_0) := s := INITIAL-STATE()
5    L(s_0) := (σ_0, waiting)
6    search(s, s_g, 1)||...||search(s, s_g, P)
7  proc update(t, τ)
8    lock(L(t))
9    for (ρ, f) ∈ L(t) do
10     if COVERS(ρ, τ)
11       unlock(L(t))
12       return true
13     else if f = waiting ∧ COVERS(τ, ρ)
14       L(t) := L(t) \ (ρ, waiting)
15   L(t) := L(t) ∪ (τ, waiting)
16   unlock(L(t))
17   return false
18 proc search((s_0, σ_0), s_g, p)
19   Q_p := if p = 1 then {(s_0, σ_0)} else ∅
20   while Q_p ≠ ∅ ∨ balance(Q_p)
21     Q_p := Q_p \ (s, σ) for some (s, σ) ∈ Q_p
22     if ¬grab(s, σ) then continue
23     for (t, τ) ∈ NEXT-STATE((s, σ)) do
24       if t = s_g then report & exit
25       if ¬update(t, τ)
26         Q_p := Q_p ∪ (t, τ)
27 proc grab(s, σ)
28   lock(L(s))
29   if σ ∉ L(s) ∨ passed = L(s, σ)
30     unlock(L(s))
31     return false
32   L(s, σ) := passed
33   unlock(L(s))
34   return true
```

Table 5.1: $\mathcal{S}$, $|\Sigma|$ ($\frac{|\Sigma|}{|\mathcal{S}|}$) and runtimes (sec) in UPPAAL and `opaal`+LTSMIN (strict BFS)

	$\lvert\mathcal{S}\rvert$	UPPAAL		`opaal`+LTSMIN (cores)						
		T	$\lvert\Sigma\rvert$	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_2	T_8	T_{16}	T_{32}	T_{48}
train-gate-N10	7e+07	837.4	1.0	1.0	1.0	573.3	297.8	76.7	39.4	21.1	14.4
viking17	1e+07	207.8	1.0	1.5	1.5	331.5	172.5	44.2	22.7	11.9	8.6
train-gate-N9	7e+06	76.8	1.0	1.0	1.0	52.4	28.5	7.7	4.1	2.4	2.0
viking15	3e+06	38.0	1.0	1.5	1.5	67.0	34.8	9.7	5.1	3.0	2.3
train-crossing	3e+04	48.3	20.8	16.1	17.3	24.5	37.2	5.8	2.7	2.0	2.1
fischer6	1e+04	0.1	0.3	50.1	50.1	219.2	129.2	46.4	36.1	32.9	31.8

Table 5.2: $|\Sigma|$ ($\frac{|\Sigma|}{|S|}$) and runtimes (sec) with non-blocking SBFS, DFS and BFS

	NB SBFS				DFS				BFS			
	$\|\Sigma_1\|$	$\|\Sigma_{48}\|$	T_1	T_{48}	$\|\Sigma_1\|$	$\|\Sigma_{48}\|$	T_1	T_{48}	$\|\Sigma_1\|$	$\|\Sigma_{48}\|$	T_1	T_{48}
train-gate-N10	1.0	1.0	547.9	14.5	1.0	1.0	647.8	15.6	1.0	1.0	559.3	13.1
viking17	1.5	1.5	320.1	9.2	1.6	1.6	386.5	9.1	1.5	1.5	325.6	7.8
train-gate-N9	1.0	1.0	52.1	2.1	1.0	1.0	61.7	1.7	1.0	1.0	51.9	1.6
viking15	1.5	1.5	64.8	2.5	1.6	1.6	80.2	3.1	1.5	1.5	66.0	2.3
train-crossing	16.1	16.1	24.1	1.8	169.8	179.0	3371.0	297.4	16.1	37.1	24.5	157.5
fischer6	50.1	50.1	201.3	12.0	54.4	39.4	405.1	10.6	50.1	58.1	206.0	32.3

Table 5.3: Memory usage (MB) of both UPPAAL (U-S0 and U-S2) and `opaal`+LTSMIN

	T	D	L	L2	Q	dbm	O+L_1	O+L_{48}	O+L_1^T	O+L_{48}^T	U-S0	U-S2
train-gate-N10	777	5989	499	499	249	1363	8101	8241	2790	3028	6091	3348
viking17	156	1040	536	214	40	87	1704	1931	828	1047	1579	722
train-gate-N9	81	549	50	50	24	61	684	815	214	347	607	332
viking15	32	190	112	44	8	55	364	581	203	423	333	162
train-crossing	0	2	5	7	0	419	426	623	425	622	48	64
fischer6	0	0	5	9	1	176	429	512	290	429	0	4

Chapter 6

Multicore Büchi Emptiness Checking for Timed Automata

This chapter is based on the paper "Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction" [66]. Section 6.7 is an independent expansion for this thesis.

The paper builds on the previous work of Chapter 5 ([43]), but expands it from reachability properties to the more general Büchi acceptance criteria which specifically includes liveness properties. One of the main goals of the paper is to exploit the partial ordering to reduce the statespace, while still preserving soundness and completeness. This results in a modified algorithm, but it could as well be viewed as a different abstraction, a view that will be expanded on in Section 6.7.

Abstract

This paper contributes to the multi-core model checking of timed automata (TA) with respect to liveness properties, by investigating checking of TA Büchi emptiness under the very coarse inclusion abstraction or zone subsumption, an open problem in this field.

We show that in general Büchi emptiness is not preserved under this abstraction, but some other structural properties are preserved. Based on those, we propose a variation of the classical nested depth-first search (NDFS) algorithm that exploits subsumption. In addition, we extend the multi-core CNDFS algorithm with subsumption, providing the first parallel LTL model checking algorithm for timed automata.

The algorithms are implemented in LTSMIN, and experimental evaluations show the effectiveness and scalability of both contributions: subsump-

tion halves the number of states in the real-world FDDI case study, and the multi-core algorithm yields speedups of up to 40 using 48 cores.

6.1 Introduction

Model checking safety properties can be done with reachability, but only guarantees that the system does not enter a dangerous state, not that the system actually serves some useful purpose. To model check such liveness properties is more involved since they state conditions over infinite executions, e.g. that a request must infinitely often produce a result. One of the most well-known logics for describing liveness properties is Linear Temporal Logic (LTL) [9].

The automata-theoretic approach for LTL model checking [96] solves the problem efficiently by translating it to the Büchi emptiness problem, which has been shown decidable for real-time systems as well [5]. However, its complexity is exponential, both in the size of the system specification and of the property. In the current paper, therefore, we consider two possible ways of alleviating this so-called state space explosion problem: (1) by utilising the many cores in modern processors, and (2) by employing coarser abstractions to the state space.

Related work. The verification of timed automata was made possible by Alur and Dill's *region construction* [5], which represents clock valuations using constraints, called *regions*. A max-clock constant abstraction, or k-extrapolation, bounded the number of regions. Since the region construction is exponential in the number of clocks and constraints in the TA, coarser abstractions such as the symbolic *zone abstraction* have been studied [47], and also implemented in, among others, the state-of-the-art model checker UPPAAL [72]. Later, the k-extrapolation for zones was refined to include lower clock constraints in the so-called lower/upper-bound (LU) abstraction proposed in [15]. Finally, the *inclusion abstraction*, or simply *subsumption*, prunes reachability according to the partial order of the symbolic states [45]. All these abstractions preserve reachability properties [45, 15].

Model checking LTL properties on timed automata, or equivalently checking timed Büchi automata (TBA) emptiness, was proven decidable in [5], by using the region construction. Bouajjani et al. [25] showed that the region-closed simulation graph preserve TBA emptiness. Tripakis [94] proved that the k-extrapolated zone simulation graph also preserves TBA emptiness, while posing the question whether other abstractions such as the LU abstraction and subsumption also preserve this property. Li [73] showed that the LU abstraction does in fact preserve TBA emptiness. The status of subsumption in LTL model checking is still open.

One way of establishing TBA emptiness on a finite simulation graph is

the nested depth-first (NDFS) algorithm [33, 60]. Recently, some multi-core version of these algorithms were introduced by Evangelista and Laarman et al [71, 50, 49]. These algorithms have the following properties: their runtime is linear in the number of states in the worst case while typically yielding good scalability; they are on-the-fly [67] and yield short counter examples [49, Sec. 4.3]. The latest version, called CNDFS, combines all these qualities and decreases memory usage [49].

In previous work, we parallelised reachability for timed automata using the mentioned abstractions [43]. It resulted in almost linear scalability, and speedups of up to 60 on a 48 core machine, compared to UPPAAL. The current work extends this previous work to the setting of liveness properties for timed automata. It also shares the UPPAAL input format, and re-uses the UPPAAL DBM library.

Problem statement. Parallel model checking of liveness properties for timed systems has been a challenge for several years. While advances were made with distributed versions of e.g. UPPAAL [12], these were limited to safety properties. Furthermore, it is unknown how subsumption, the coarsest abstraction, can be used for checking TBA emptiness.

Contributions. (1) For the first time, we realize parallel LTL model checking of timed systems using the CNDFS algorithm. (2) We prove that subsumption preserves several structural state space properties (Section 6.3), and show how these properties can be exploited by NDFS and CNDFS (Section 6.4 and Section 6.5). (3) We implement NDFS and CNDFS with subsumption in the LTSMIN toolset [68] and `opaal` [42]. Finally, (4) our experiments show considerable state space reductions by subsumption and good parallel scalability of CNDFS with speedups of up to 40 using 48 cores.

6.2 Preliminaries: Timed Büchi Automata and Abstractions

In the current section, we first recall the formalism of timed Büchi automata (TBA), that allows modelling of both a real-time system and its liveness requirements. Subsequently, we introduce finite symbolic semantics using zone abstraction with extrapolation and subsumption. Finally, we show which properties are known to be preserved under said abstractions.

Timed Automata and Transition Systems. Definition 36 provides a basic definition of a TBA. It can be extended with features such as finitely valued variables, and parallel composition to model networks of timed automata, as done in UPPAAL [16].

Definition 35 (Guards)**.** *Let $\mathcal{G}(\mathcal{C})$ be a conjunction of clock constraints over the set of clocks $c \in \mathcal{C}$, generalized by:*

$$g ::= c \bowtie n \mid g \wedge g \mid true$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, =, >, \geq\}$ is a comparison operator. We call a guard downwards closed if all $\bowtie \in \{<, \leq, =\}$.

Definition 36 (Timed Büchi Automaton)**.** *A timed Büchi automaton (TBA) is a 6-tuple $\mathbb{B} = (L, \mathcal{C}, \mathcal{F}, l_0, \rightarrow, I_{\mathcal{C}})$, where*

- *L is a finite set of locations, typically denoted by ℓ, where $\ell_0 \in L$ is the initial location, and $\mathcal{F} \subseteq L$, is the set of accepting locations,*
- *$\mathcal{C}$ is a finite set of clocks, typically denoted by c,*
- *$\rightarrow \subseteq L \times \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is the (non-deterministic) transition relation. We write $\ell \xrightarrow{g,R} \ell'$ for a transition, where ℓ is the source and ℓ' the target location, $g \in \mathcal{G}(\mathcal{C})$ is a transition guard, $R \subseteq \mathcal{C}$ is the set of clocks to reset, and*
- *$I_{\mathcal{C}} \colon L \rightarrow \mathcal{G}(\mathcal{C})$ is an invariant function, mapping locations to a set of guards. To simplify the semantics, we require invariants to be downwards-closed.*

The states of a TBA involve the notion of clock valuations. A clock valuation is a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$. We denote all clock valuations over $\mathcal{C}$ with $\mathcal{V}_{\mathcal{C}}$. We need two operations on clock valuations: $v' = v + \delta$ for a delay of $\delta \in \mathbb{R}_{\geq 0}$ time units s.t. $\forall c \in \mathcal{C} \colon v'(c) = v(c) + \delta$, and reset $v' = v[R]$ of a set of clocks $R \subseteq \mathcal{C}$ s.t. $v'(c) = 0$ if $c \in R$, and $v'(c) = v(c)$ otherwise. We write $v \models g$ to mean that the clock valuation v satisfies the clock constraint g.

Definition 37 (Transition system semantics of a TBA)**.** *The semantics of a TBA $\mathbb{B}$ is defined over the transition system $\mathcal{TS}_v^{\mathbb{B}} = (\mathcal{S}_v, s_0, \mathcal{T}_v)$ s.t.:*

1. *A state $s \in \mathcal{S}_v$ is a pair: (ℓ, v) with a location $\ell \in L$, and a clock valuation v.*
2. *An initial state $s_0 \in \mathcal{S}_v$, s.t. $s_0 = (\ell_0, v_0)$, where $\forall c \in \mathcal{C} \colon v_0(c) = 0$.*
3. *$\mathcal{T}_v \colon \mathcal{S}_v \times (\{\epsilon\} \cup \mathbb{R}_{\geq 0}) \times \mathcal{S}_v$ is a transition relation with $(s, a, s') \in \mathcal{T}_v$, denoted $s \xrightarrow{a} s'$ s.t. there are two types of transitions:*
   (a) *A discrete (instantaneous) transition: $(\ell, v) \xrightarrow{\epsilon} (\ell', v')$ if an edge $\ell \xrightarrow{g,R} \ell'$ exists, $v \models g$ and $v' = v[R]$, and $v' \models I_{\mathcal{C}}(\ell')$.*
   (b) *A delay by δ time units: $(\ell, v) \xrightarrow{\delta} (\ell, v + \delta)$ for $\delta \in \mathbb{R}_{\geq 0}$ if $v + \delta \models I_{\mathcal{C}}(\ell)$.*

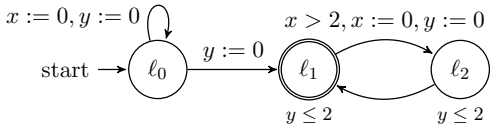


Figure 6.1: A timed Büchi automaton.

We say a state $s \in \mathcal{S}$ is accepting, or $s \in \mathcal{F}$, when $s = (\ell, \ldots)$ and $\ell \in \mathcal{F}$. We write $s \xrightarrow{\delta}\xrightarrow{\epsilon} s'$ if there exists a state s'' such that $s \xrightarrow{\delta} s''$ and $s'' \xrightarrow{\epsilon} s'$. We denote an infinite run in $\mathcal{TS}_v^{\mathbb{B}} = (\mathcal{S}_v, s_0, \mathcal{T}_v)$ as an infinite path $\pi = s_1 \xrightarrow{\delta_1}\xrightarrow{\epsilon} s_2 \xrightarrow{\delta_2}\xrightarrow{\epsilon} s_3 \ldots$ The run is accepting if there exist an infinite number of indices i s.t. $s_i \in \mathcal{F}$. A(n infinite) run's time lapse is $Time(\pi) = \sum_{i \geq 1} \delta_i$. An infinite path π in $\mathcal{TS}_v^{\mathbb{B}}$ is *time convergent*, or zeno, if $Time(\pi) < \infty$, otherwise it is divergent. For example, the TBA in Figure 6.1 has an infinite run: $(\ell_0, v_0) \xrightarrow{1} (\ell_0, v_0) \xrightarrow{1} \cdots$, that is not accepting, but is non-zeno. We claim that there is no accepting non-zeno run, exemplified by the finite run: $(\ell_0, v_0) \xrightarrow{2}\xrightarrow{\epsilon} (\ell_1, v_1) \xrightarrow{0}\xrightarrow{\epsilon} (\ell_2, v_0) \xrightarrow{0}\xrightarrow{\epsilon} (\ell_1, v_0) \xrightarrow{1.9}\not\xrightarrow{\epsilon}$.

Definition 38 (A TBA's language and the emptiness problem)**.** *The language accepted by $\mathbb{B}$, denoted $\mathcal{L}(\mathbb{B})$, is defined as the set of non-zeno accepting runs. The language emptiness problem for $\mathbb{B}$ is to check whether $\mathcal{L}(\mathbb{B}) = \emptyset$.*

Remark 1 (Zenoness)**.** *Zenoness is considered a modelling artifact as the behaviour it models cannot occur in any real system, which after all has finite processing speeds. Therefore, zeno runs should be excluded from analysis. However, any TBA $\mathbb{B}$ can be syntactically transformed to a strongly non-zeno $\mathbb{B}$' [95], s.t. $\mathcal{L}(\mathbb{B}) = \emptyset$ iff $\mathcal{L}(\mathbb{B}') = \emptyset$. Therefore, in the following, w.l.o.g., we assume that all TBAs are strongly non-zeno.*

Definition 39 (Time-abstracting simulation relation)**.** *A time-abstracting simulation relation R is a binary relation on $\mathcal{S}_v$ s.t. if $s_1 R s_2$ then:*

- *If $s_1 \xrightarrow{\epsilon} s_1'$, then there exists s_2' s.t. $s_2 \xrightarrow{\epsilon} s_2'$ and $s_1' R s_2'$.*
- *If $s_1 \xrightarrow{\delta} s_1'$, then there exists s_2' and δ' s.t. $s_2 \xrightarrow{\delta'} s_2'$ and $s_1' R s_2'$.*

If both R and R^{-1} are time-abstracting simulation relations, we call R a time-abstracting bisimulation relation.

Symbolic Abstractions using Zones. A *zone* is a symbolic representation of an infinite set of clock valuations by means of a clock constraint. These constraints are conjuncts (Definition 40) of simple linear inequalities on clock values, and thus describe (unbounded) convex polytopes in a $|\mathcal{C}|$-dimensional plane (e.g. Figure 6.2). Therefore, zones can be efficiently represented by Difference Bounded Matrices (DBMs) [19].

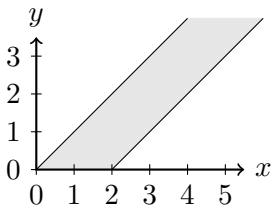


Figure 6.2: A graphical representation of a zone over 2 clocks: $0 \leq x - y \leq 2$.

Definition 40 (Zones)**.** *Similar to the guard definition, let $\mathcal{Z}(\mathcal{C})$ be the set of clock constraints over the set of clocks $c, c_1, c_2 \in \mathcal{C}$ generalized by:*

$$Z ::= c \bowtie n \mid c_1 - c_2 \bowtie n \mid Z \wedge Z \mid \mathit{true} \mid \mathit{false}$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, >, \geq\}$ is a comparison operator. We also use $=$ for equalities, short for the conjunction of $\leq$ and $\geq$.

We write $v \models Z$ if the clock valuation v is included in Z, for the set of clock valuations in a zone $[\![Z]\!] = \{v \mid v \models Z\}$, and for *zone inclusion* $Z \subseteq Z'$ iff $[\![Z]\!] \subseteq [\![Z']\!]$. Notice that $[\![\mathit{false}]\!] = \emptyset$. Using the fundamental operations below, which are detailed in [19], we define the *zone semantics* over *simulation graphs* in Definition 41. Most importantly, these operations are implementable in $O(n^3)$ or $O(n^2)$ and closed w.r.t. $\mathcal{Z}$.

***delay*:** $[\![Z \uparrow]\!] = \{v + \delta | \delta \in \mathbb{R}_{\geq 0}, v \in [\![Z]\!]\}$,

***clock reset*:** $[\![Z[R]]\!] = \{v[R] | v \in [\![Z]\!]\}$, and

***constraining*:** $[\![Z \wedge Z']\!] = [\![Z]\!] \cap [\![Z']\!]$.

Definition 41 (Zone semantics)**.** *The semantics of a TBA $\mathbb{B} = (L, \mathcal{C}, \mathcal{F}, \ell_0, \rightarrow, I_{\mathcal{C}})$ under the zone abstraction is a simulation graph: $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ s.t.:*

1. *$\mathcal{S}_{\mathcal{Z}}$ consists of pairs (ℓ, Z) where $\ell \in L$, and $Z \in \mathcal{Z}$ is a zone.*
2. *$s_0 \in \mathcal{S}_{\mathcal{Z}}$ is an initial state $(\ell_0, Z_0 \uparrow \wedge\ I_{\mathcal{C}}(\ell_0))$ with $Z_0 = \bigwedge_{c \in \mathcal{C}} c = 0$.*
3. *$\mathcal{T}_{\mathcal{Z}}$ is the symbolic transition function using zones, s.t. $(s, s') \in \mathcal{T}_{\mathcal{Z}}$, denoted $s \Rightarrow s'$ with $s = (\ell, Z)$ and $s' = (\ell', Z')$, if an edge $\ell \xrightarrow{g,R} \ell'$ exists, and $Z \wedge g \neq \mathit{false}$, $Z' = (((Z \wedge g)[R]) \uparrow) \wedge\ I_{\mathcal{C}}(\ell')$ and $Z' \neq \mathit{false}$.*

Any simulation graph is a discrete graph, hence cycles and lassos are defined in the standard way. We write $s \Rightarrow^+ s'$ iff there is a non-empty path in $SG(\mathbb{B})$ from s to s', or $s \Rightarrow^* s'$ if the path can be empty. An infinite run in $SG(\mathbb{B})$ is an infinite sequence of states $\pi = s_1 s_2 \ldots$, s.t. $s_i \Rightarrow s_{i+1}$ for all $i \geq 1$. It is accepting if it contains infinitely many accepting states. If $SG(\mathbb{B})$ is finite, any infinite path from s_0 defines a lasso: $s_0 \Rightarrow^* s \Rightarrow^+ s$.

Definition 42 (A TBA's language under Zone Semantics)**.** *The language accepted by a TBA $\mathbb{B}$ under the zone semantics, denoted $\mathcal{L}(SG(\mathbb{B}))$, is the set of infinite runs $\pi = s_0 s_1 s_2 \ldots$ s.t. there exists an infinite set of indices s.t. $s_i \in \mathcal{F}$.*

Because there are infinitely many zones, the state space of $SG(\mathbb{B})$ may also be infinite. To bound the number of zones, *extrapolation* methods combine all zones which a given TBA $\mathbb{B}$ cannot distinguish. For example, k-extrapolation finds the largest upper bound k in the guards and invariants of $\mathbb{B}$, and extrapolates all bounds in the zones $\mathcal{Z}$ that exceed this value, while LU-extrapolation uses both the maximal lower bound l and the maximal upper bound u [15]. Extrapolation can be refined on a per-clock basis [15], and on a per-location basis.

Definition 43. *An abstraction over a simulation graph $SG(\mathbb{B}) = (\mathcal{S}_\mathcal{Z}, s_0, \mathcal{T}_\mathcal{Z})$ is a mapping $\alpha : \mathcal{S}_\mathcal{Z} \to \mathcal{S}_\mathcal{Z}$ s.t. if $\alpha((\ell, Z)) = (\ell', Z')$ then $\ell = \ell'$ and $Z \subseteq Z'$. If the image of an abstraction α is finite, we call it a finite abstraction.*

Definition 44. *Abstraction α over zone transition system $SG(\mathbb{B}) = (\mathcal{S}_\mathcal{Z}, s_0, \mathcal{T}_\mathcal{Z})$ induces a zone transition system $SG_\alpha(\mathbb{B}) = (\mathcal{S}_\alpha, \alpha(s_0), \mathcal{T}_\alpha)$ where:*

- *$\mathcal{S}_\alpha = \{\alpha(s) \mid s \in \mathcal{S}_\mathcal{Z}\}$ is the set of states, s.t. $\mathcal{S}_\alpha \subseteq \mathcal{S}_\mathcal{Z}$,*
- *$\alpha(s_0)$ is the initial state, and*
- *$(s, s') \in \mathcal{T}_\alpha$ iff $(s, s'') \in \mathcal{T}_\mathcal{Z}$ and $s' = \alpha(s'')$, is the transition relation.*

We call an abstraction α an *extrapolation* if there exists a simulation relation R s.t. if $\alpha((\ell, Z)) = (\ell, Z')$ then for all $v' \in Z'$ there exist a $v \in Z$ s.t. $v' R v$ [73]. This means extrapolations do not introduce behaviour that the un-extrapolated system cannot simulate. The abstraction defined by k-extrapolation is denoted by α_k, while the abstraction defined by LU-extrapolation is called α_{lu}. Hence, α_k and α_{lu} induce finite simulation graphs, written $SG_k(\mathbb{B})$ and $SG_{lu}(\mathbb{B})$.

Subsumption abstraction. While $SG_k(\mathbb{B})$ and $SG_{lu}(\mathbb{B})$ are finite, their size is still exponential in the number of clocks. Therefore, we turn to the coarser inclusion/ subsumption abstraction of [45], hereafter denoted *subsumption abstraction*. We extend the notion of subsumption to states: a state $s = (\ell, Z) \in \mathcal{S}_\mathcal{Z}$ is *subsumed* by another $s' = (\ell', Z')$, denoted $s \sqsubseteq s'$, when $\ell = \ell'$ and $Z \subseteq Z'$. Let $\mathcal{R}(SG(\mathbb{B})) = \{s | s_0 \Rightarrow^* s\}$ denote the set of *reachable states* in $SG(\mathbb{B})$.

Proposition 1 ($\sqsubseteq$ is a simulation relation)**.** *If $(\ell, Z_1) \sqsubseteq (\ell, Z_2)$ and $(\ell, Z_1) \Rightarrow (\ell', Z_1')$ then there exists Z_2' s.t. $(\ell, Z_2) \Rightarrow (\ell', Z_2')$ and $(\ell', Z_1') \sqsubseteq (\ell', Z_2')$.*

Proof. By the definition of $\sqsubseteq$, and the fact that $\Rightarrow$ is monotone w.r.t $\subseteq$ of zones. □

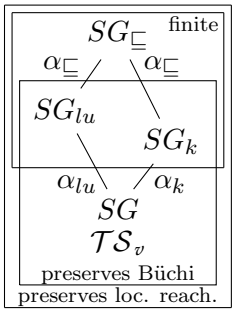


Figure 6.3: Abstractions.

Definition 45 (Subsumption abstraction [45])**.** *A subsumption abstraction $\alpha_{\sqsubseteq}$ over a zone transition system $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ is a total function $\alpha_{\sqsubseteq} : \mathcal{R}(SG(\mathbb{B})) \to \mathcal{R}(SG(\mathbb{B}))$ s.t. $s \sqsubseteq \alpha_{\sqsubseteq}(s)$*

Note the subsumption abstraction is defined only over the reachable state space, and is *not* an extrapolation, because it might introduce extra transitions that the unabstracted system cannot simulate. Typically α is constructed on-the-fly during analysis, only abstracting to states that are already found to be reachable. This makes its performance depend heavily on the search order, as finding "large" states quickly can make the abstraction coarser [43].

Property preservation under abstractions. We now consider the preservation by the abstractions above of the property of *location reachability* (a location ℓ is reachable iff $s_0 \Rightarrow^* (\ell, \ldots)$) and that of Büchi emptiness.

Proposition 2. *For any of the abstractions α: α_k [45], α_{lu} [15], $\alpha_k \circ \alpha_{\sqsubseteq}$ [45], and $\alpha_{lu} \circ \alpha_{\sqsubseteq}$ [15], it holds that*

$$\ell \text{ is reachable in } \mathcal{TS}_v^{\mathbb{B}} \iff \ell \text{ is reachable in } SG_\alpha(\mathbb{B})$$

Proposition 3. *For any finite extrapolation [73] α, e.g. the abstractions α_k [94] and α_{lu} [73] it holds that*

$$\mathcal{L}(\mathbb{B}) = \emptyset \iff \mathcal{L}(SG_\alpha(\mathbb{B})) = \emptyset$$

From hereon we will denote any finite extrapolation as α_{fin}, and the associated simulation graph $SG_{fin}(\mathbb{B})$. To denote that this graph can be generated *on-the-fly* [96, 9, 45], we use a NEXT-STATE(s) function which returns the set of successor states for s: $\{s' \in \mathcal{S}_{fin} \mid s \Rightarrow s'\}$.

As a result of Proposition 3 we can focus on finding accepting runs in $SG_{fin}(\mathbb{B})$. Because it is finite, any such run is represented by a lasso: $s_0 \Rightarrow s \Rightarrow^+ s$. Tripakis [94] poses the question of whether $\alpha_{\sqsubseteq}$ can be used to check Büchi emptiness. We will investigate this further in the next section.

6.3 Preservation of Büchi Emptiness under Subsumption

The current section, investigates what properties are preserved by a subsumption abstraction $\alpha_{\sqsubseteq}$, when applied on a finite simulation graph obtained by an extrapolation, α_{fin}, in the following, denoted as $SG_{\sqsubseteq}(\mathbb{B}) = (SG_{\mathit{fin}\circ\sqsubseteq}(\mathbb{B}))$.

Proposition 4. *For all abstractions α, $s \in \mathcal{F} \Leftrightarrow \alpha(s) \in \mathcal{F}$ (by Definition 43).*

Proposition 5. *An $\alpha_{\sqsubseteq}$ abstraction is safe w.r.t. Büchi emptiness:*

$$\mathcal{L}(\mathbb{B}) \neq \emptyset \implies \mathcal{L}(SG_{\sqsubseteq}(\mathbb{B})) \neq \emptyset$$

Proof. If $\mathcal{L}(\mathbb{B}) \neq \emptyset$, there must be an infinite accepting path π. This path is inscribed [94] in $SG_{\mathit{fin}}(\mathbb{B})$, and because $\sqsubseteq$ is a simulation relation a similar path exists in $SG_{\sqsubseteq}(\mathbb{B})$. □

Proposition 5 shows that subsumption abstraction preserves Büchi emptiness in one direction. Unfortunately, an accepting cycle in $SG_{\sqsubseteq}(\mathbb{B})$ is not always reflected in $SG_{\mathit{fin}}(\mathbb{B})$, as Figure 6.4 illustrates. The figure visualises $SG_{\sqsubseteq}(\mathbb{B})$ by drawing subsumed states inside subsuming states (e.g. $s_3 \sqsubseteq s_1$).

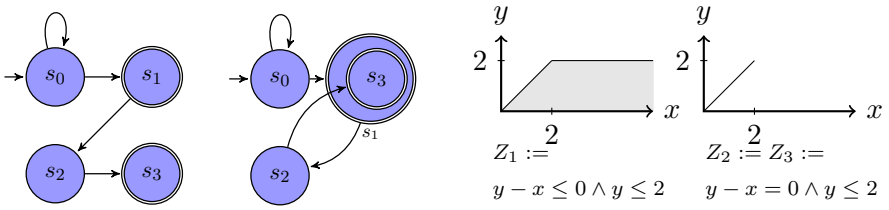


Figure 6.4: The state space $SG_{\sqsubseteq}(\mathbb{B})$ of the model in Figure 6.1 with $\ell_1 \in \mathcal{F}$ contains 4 states (shown on the left): s_0, $s_1 = (\ell_1, Z_1)$, $s_2 = (\ell_2, Z_2)$ and $s_3 = (\ell_1, Z_3)$. The graphical representation of the zones Z_1–Z_3 (right) reveals that $Z_3 \subseteq Z_1$ and hence $s_3 \sqsubseteq s_1$. As $s_3 \sqsubseteq s_1$ and both are reachable, a subsumption abstraction is allowed to map $\alpha_{\sqsubseteq}(s_3) = s_1$, introducing a cycle $s_1 \Rightarrow s_2 \Rightarrow s_1$ in $SG_{\sqsubseteq}(\mathbb{B})$.

However, subsumption introduces strong properties on paths and cycles to which we devote the rest of the current section. In subsequent sections, we exploit these properties to improve algorithms that implement the TBA emptiness check.

Lemma 7 (Accepting cycles under $\sqsubseteq$)**.** *If $SG_{\mathit{fin}}(\mathbb{B})$ contains states s, s' s.t. s leads to an accepting cycle and $s \sqsubseteq s'$, then s' leads to an accepting cycle.*

Proof. We have that $s \Rightarrow^* t \Rightarrow^+ t$, and because $\sqsubseteq$ is a simulation relation we have the existence of a state x s.t. $t \sqsubseteq x$:

$$\begin{array}{ccccccc} s' & \Rightarrow^* & t' & \Rightarrow & \cdots & \Rightarrow & x \\ \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| \\ s & \Rightarrow^* & t & \Rightarrow & \cdots & \Rightarrow & t \end{array}$$

From x, we again have a similar path, to some x'. This sequence will eventually repeat some x'', because $SG_{fin}(\mathbb{B})$ is finite. It follows that all states in $x'' \Rightarrow^+ x''$ subsume states in $t \Rightarrow^+ t$, hence the former cycle is also accepting (Proposition 4). □

Lemma 8 (Paths under $\sqsubseteq$)**.** *If $SG_{fin}(\mathbb{B})$ contains a path $s \Rightarrow^+ s'$ containing an accepting state and $s \sqsubseteq s'$, then s leads to an accepting cycle.*

Proof. Because $\sqsubseteq$ is a simulation relation we have that $s \Rightarrow^+ s'$ and $s \sqsubseteq s'$ implies the existence of some t such that $s' \Rightarrow^+ t$ and $s' \sqsubseteq t$. From t, we again obtain a similar path to some t', s.t. $t \sqsubseteq t'$. Because $SG_{fin}(\mathbb{B})$ is finite, the sequence of t's will eventually repeat some element x, s.t. $x \Rightarrow^+ \cdots \Rightarrow^+ x$.

$$\begin{array}{ccccccccccc} s' & \Rightarrow^+ & t & \Rightarrow^+ & t' & \Rightarrow^+ & \cdots & \Rightarrow^+ & t'' & \Rightarrow^+ & x \\ \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \| \\ s & \Rightarrow^+ & s' & \Rightarrow^+ & t & \Rightarrow^+ & \cdots & \Rightarrow^+ & x & \Rightarrow^+ & x \end{array}$$

This gives us the lasso $s \Rightarrow^* x \Rightarrow^+ x$. It also follows that all states in $x \Rightarrow^+ x$ subsume states in $s \Rightarrow^+ s'$, hence the former cycle is accepting (Proposition 4). □

6.4 Timed Nested Depth-First Search with Subsumption

Algorithm 10 NDFS

```
 1: procedure ndfs( )
 2:     Cyan := Blue := Red := ∅
 3:     dfsBlue(s_0)
 4:     report no cycle
 5: procedure dfsRed(s)
 6:     Red := Red ∪ {s}
 7:     for all t in NEXT-STATE(s) do
 8:         if t ∈ Cyan then
 9:             report cycle
10:         if (t ∉ Red) then dfsRed(t)
11: procedure dfsBlue(s)
12:     Cyan := Cyan ∪ {s}
13:     for all t in NEXT-STATE(s) do
14:         if t ∉ Blue ∧ t ∉ Cyan then
15:             dfsBlue(t)
16:     if s ∈ F then
17:         dfsRed(s)
18:     Blue := Blue ∪ {s}
19:     Cyan := Cyan \ {s}
```

In the current section, we extend the classic linear-time NDFS [33, 93] algorithm to exploit subsumption. The algorithm detects accepting cycles, the absence of which implies Büchi emptiness. It is correct for the graph $SG_{fin}(\mathbb{B})$ according to Proposition 3. In the following, with *soundness*, we mean that when NDFS reports a cycle, indeed an accepting cycle exists in the graph, while completeness indicates that NDFS always reports an accepting cycle if the graph contains one.

The NDFS algorithm in Algorithm 10 consists of an outer DFS (*dfsBlue*) that sorts accepting states s in DFS *postorder*. And an inner DFS (*dfsRed*) that searches for cycles over each s, called the *seed*. States are maintained in 3 colour sets:

1. *Blue*, states *explored* by *dfsBlue*,
2. *Cyan*, states on the stack of *dfsBlue* (*visited* but not yet explored), which are used by *dfsRed* to close cycles over s early at l.8 [93], and
3. *Red*, visited by *dfsRed*.

Algorithm 10 maintains a few strong invariants, which are already mentioned in [33, 93]:

I0: At l.14 all red states are blue.

I1: The only accepting state visited by *dfsRed* is the seed.

I2: Outside of *dfsRed*, accepting cycles are not reachable from red states.

I3: A sufficient postcondition for *dfsRed*(s) is that all reachable states from s are included in *Red* and no cyan state is reachable from it.

We now try to employ subsumption on the different colours to prune the searches, even though we cannot use it on all colours as $SG_{\sqsubseteq}(\mathbb{B})$ introduces additional cycles as Figure 6.4 showed. To express subsumption checks on sets we write $s \sqsubseteq S$, meaning $\exists s' \in S\colon s \sqsubseteq s'$. And $S \sqsubseteq s$, meaning $\exists s' \in S\colon s' \sqsubseteq s$. At several places in Algorithm 10 we might apply subsumption, leading to the following options:

1. On cyan for cycle detection:
   (a) $t \sqsubseteq \mathit{Cyan}$ at l.8, or
   (b) $\mathit{Cyan} \sqsubseteq t$ at l.8.
2. On *dfsBlue*, by replacing $t \notin \mathit{Blue} \wedge t \notin \mathit{Cyan}$ at l.14 with $t \not\sqsubseteq \mathit{Blue} \cup \mathit{Cyan}$.
3. On the blue set (explored states), by replacing $t \notin \mathit{Blue}$ at l.14 with $t \not\sqsubseteq \mathit{Blue}$.

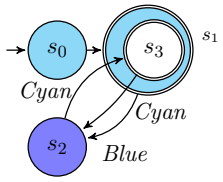


Figure 6.5: Counter example to subsumption on *Blue* and *Cyan* (item 2).

4. On *dfsRed*, by replacing $t \notin Red$ at l.10 with $t \not\sqsubseteq Red$.

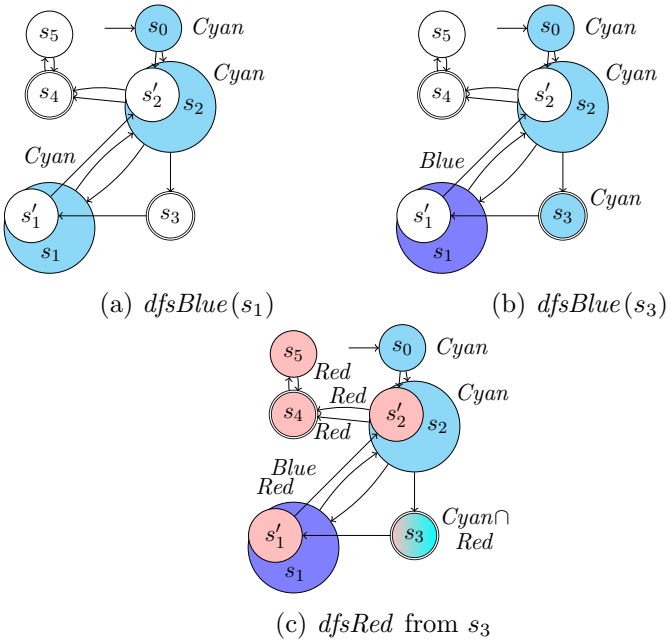
Subsumption on cyan for cycle detection as in item 1a makes the algorithm unsound: cycles in $SG_{\sqsubseteq}(\mathbb{B})$ are not always reflected in $SG_{fin}(\mathbb{B})$ (Figure 6.4). There is also no hope of "unwinding" the algorithm upon detecting an accepting cycle that does not exist in the underlying $SG_{fin}(\mathbb{B})$ without losing its linear-time complexity, as the number of cycles can be exponential in the size of $SG_{\sqsubseteq}(\mathbb{B})$.

If, on the other hand, we prune the blue search as in item 2, the algorithm becomes incomplete. Figure 6.5 shows a run of the modified NDFS on an $SG_{fin}(\mathbb{B})$ with cycle $s_3 \Rightarrow s_2 \Rightarrow s_3$. The *dfsBlue* backtracked over s_2 as $s_3 \sqsubseteq s_1$ and $s_1 \in Cyan$. The *dfsRed* now launched from s_1, will however continue to visit s_3, while missing the cycle as s_2 is not cyan. We also observe that I1 is violated, indicating that the postorder on accepting states (s_3 before s_1) is lost.

It is tempting therefore to use subsumption on blue only, as in item 3. However, Figure 6.6 shows an "animation" of a run with the modified NDFS which is incomplete. Here state s_1 is first backtracked in the blue search as all successors are cyan (left). Then state s_1 is marked blue; The blue search backtracks to s_2, proceeds to s_3 and backtracks because it finds $s_1' \sqsubseteq s_1 \in Blue$ (middle). Then a red search is started from s_3, which subsumes the cyan stack (s_2) and visits accepting state s_4, violating I1 and missing the accepting cycle $s_4 \Rightarrow s_5 \Rightarrow s_4$.

A viable option however is to use inverse subsumption on cyan as in item 1b. According to Lemma 7, a state that subsumes a state on the cyan stack leads to a cycle. And as the only goal of the red search is to find a cyan state (to close an accepting cycle over the seed), it does not rely on DFS (I3). Thus we may as well use subsumption in the red search as in item 4. By definition (Definition 45), $SG_{\sqsubseteq}(\mathbb{B})$ contains a "larger" state for all reachable states in $SG_{fin}(\mathbb{B})$. So in combination with item 1b this is sufficient to find all accepting cycles.

The strong invariant (I2) states accepting cycles are not reachable from red states, so red states can prune the blue search. We can strengthen the condition on l.14 to $t \notin Blue \cup Cyan \cup Red$. However, this is of no use since by (I0), $Red \subseteq Blue$. Luckily, even states subsumed by red do not lead to



(a) *dfsBlue*(s_1) (b) *dfsBlue*(s_3)

(c) *dfsRed* from s_3

Figure 6.6: Counter example to subsumption on *Blue*

accepting cycles (contraposition of Lemma 7), so we can use subsumption again: $t \notin \mathit{Blue} \cup \mathit{Cyan} \wedge t \not\sqsubseteq \mathit{Red}$. The benefit of this can be illustrated using Figure 6.4. Once *dfsBlue* backtracks over s_1, we have $s_1, s_2, s_3 \in \mathit{Red}$ by *dfsRed* at l.17. Any hypothetical other path from s_0 to a state subsumed by these red states can be ignored.

Algorithm 11 shows a version of NDFS with all correct improvements. Notice that I2 and I3 are sufficient to conclude correctness of these modifications.

6.5 Multi-Core CNDFS with Subsumption

CNDFS [49] is a parallel algorithm for checking Büchi emptiness [49]. By Proposition 3, it is correct for SG_{fin}. In the current section, we extend CNDFS with subsumption, in a similar way as we have done for the sequential NDFS in the previous section.

In CNDFS (Algorithm 12 without underlined parts), each worker thread i runs a seemingly independent $\mathit{dfsBlue}_i$ and dfsRed_i, with a local stack colour Cyan_i, and its own random successor ordering (indicated by the subscript i of the NEXT-STATE function). However, the workers assist each other by sharing the colours *Blue* and *Red* globally, thus pruning each other's search space.

The main problem that CNDFS has to solve is the loss of postorder on

Algorithm 11 NDFS with subsumption on red, cycle detection, and red prune of *dfsBlue*.

```
1:  procedure ndfs( )
2:     Cyan := Blue := Red := ∅
3:     dfsBlue(s_0)
4:     report no cycle
5:  procedure dfsRed(s)
6:     Red := Red ∪ {s}
7:     for all t in NEXT-STATE(s) do
8:        if Cyan ⊑ t then
9:           report cycle
10:       if (t ⋢ Red) then dfsRed(t)
11: procedure dfsBlue(s)
12:    Cyan := Cyan ∪ {s}
13:    for all t in NEXT-STATE(s) do
14:       if (t ∉ Blue ∪ Cyan ∧ t ⋢ Red)
15:       then dfsBlue(t)
16:    if s ∈ F then
17:       dfsRed(s)
18:    Blue := Blue ∪ {s}
19:    Cyan := Cyan \ {s}
```

the accepting states due to the shared blue color (similar to the effects of item 3 as illustrated in Figure 6.6). In the previous section, we have seen that a loss of postorder may cause *dfsRed* to visit non-seed accepting states, i.e. I1 is violated. CNDFS demonstrates that repairing the latter *dangerous situation* is sufficient to preserve correctness [49, Sec. 3].

To detect a dangerous situation, CNDFS collects the states visited by $dfsRed_i$ in a set $\mathcal{R}_i$ at l.7. After completion of $dfsRed_i$, the algorithm then checks $\mathcal{R}_i$ for non-seed accepting states at l.21. By simply waiting for these states to become red, the dangerous situation is resolved as the blue state that caused the situation was always placed by some other worker, which will eventually continue [49, Prop. 3]. Once the situation is detected to be resolved, all states from the local $\mathcal{R}_i$ are added to *Red* at l.22.

CNDFS maintains similar invariants as NDFS:

I2' Red states do not lead to accepting cycles (Lemma 1 and Prop. 2 in [49]).

I3' After $dfsRed_i(s)$ states reachable from s are red or in $\mathcal{R}_i$ (from [49, Lem. 2]).

Because these invariants are as strong or stronger than I2 and I3, we can use subsumption in a similar way as for NDFS. Algorithm 12 underlines the changes to algorithm w.r.t. Alg. 2 in [49]. We additionally have to extend the waiting procedure to include subsumption at l.21, because the use of subsumption in $dfsRed_i$ can cause other workers to find "larger" states.

In the next section, we will benchmark Algorithm 12 on timed models. An important property that the algorithm inherits from CNDFS, is that its *runtime* is linear in the size of the input graph N. However, in the worst case, all workers may visit the same states. Therefore, the complexity of the amount of *work* that the algorithm performs (or the amount of power it

Algorithm 12 CNDFS with subsumption

```
1:  procedure cndfs(P)
2:      Blue := Red := ∅
3:      forall i in 1..P do Cyan_i := ∅
4:      dfsBlue_1(s_0) ‖ .. ‖ dfsBlue_P(s_0)
5:      report no cycle
6:  procedure dfsRed_i(s)
7:      R_i := R_i ∪ {s}
8:      for all t in NEXT-STATE_i(s) do
9:          if Cyan ⊑ t then cycle
10:         if t ∉ R_i ∧ t ⋤ Red then
11:             dfsRed_i(t)
12: procedure dfsBlue_i(s)
13:     Cyan_i := Cyan_i ∪ {s}
14:     for all t in NEXT-STATE_i(s) do
15:         if t ∉ Cyan_i ∪ Blue ∧ t ⋤ Red then
16:             dfsBlue(t)
17:     Blue := Blue ∪ {s}
18:     if s ∈ F then
19:         R_i := ∅
20:         dfsRed(s)
21:         await ∀s' ∈ R_i ∩ F \ {s}: s' ⊑ Red
22:         forall s' in R_i do Red := Red ∪ s'
23:     Cyan_i := Cyan_i \ {s}
```

consumes) equals $N \cdot P$, where P is the number of processors used. The randomised successor function NEXT-STATE$_i$ however ensures that this does not happen for most practical inputs. Experiments on over 300 examples confirmed this [49, Sec. 4], making CNDFS the current state-of-the-art parallel LTL model checking algorithm.

6.6 Experimental Evaluation

To evaluate the performance of the proposed algorithms experimentally, we implemented CNDFS without [49] and with subsumption (Algorithm 12) in LTSMIN 2.0[1]. The `opaal` [42] tool[2] functions as a front-end for UPPAAL models. Previously, we demonstrated scalable multi-core reachability for timed automata [43].

Experimental setup. We benchmarked[3] on a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 6168) with a varying number of threads, averaging results over 5 repetitions. We consider the following models and LTL properties:

`csma`[4] is a protocol for Carrier Sense, Multiple-Access with Collision Detection with 10 nodes. We verify the property that on collisions, eventually the bus will be active again: $\Box$`((P0=bus_collision1)` $\Longrightarrow$ $\Diamond$`(P0=bus_active))`.

`fischer-1/2`[5] implements a mutual exclusion protocol; a canonical benchmark for timed automata, with 10 nodes. As in [73], we use the property (1): $\neg$`((`$\Box\Diamond$`k=1)`$\vee$`(`$\Box\Diamond$`k=0))`, where `k` is the number of processes in their critical section. We also add a weak fairness property (2): $\Box$`((`$\Box$`P_1=req)` $\Longrightarrow$ `(`$\Diamond$`P_1=cs))`: processes requesting infinitely often will eventually be served.

`fddi`[4] models a token ring system as described in [25], where a network of 10 stations are organised in a ring and can hand back the token in a synchronous or asynchronous fashion. We verify the property from [25] that every station will eventually send asynchronous messages: $\Box$`(`$\Diamond$`(ST1=station_z_sync))`.

`train-gate`[4] models a railway interlocking, with 10 trains. Trains drive onto the interconnect until detected by sensors. There they wait until receiving a signal for safe crossing. The property prescribes that each

[1] Available as open source at: `http://fmt.cs.utwente.nl/tools/ltsmin`
[2] Available as open source at: `http://opaal-modelchecker.com`
[3] All results are available at: `http://fmt.cs.utwente.nl/tools/ltsmin/cav-2013`
[4] From `http://www.it.uu.se/research/group/darts/uppaal/benchmarks/`
[5] As distributed with UPPAAL.

approaching train eventually should be serviced: $\Box$(`Train_1`=`Appr` $\Longrightarrow$ ($\Diamond$`Train_1`=`Cross`)).

The following command-line was used to start the LTSMIN tool: opaal2lts-mc --strategy=[A] --ltl-semantics=textbook --ltl=[f] -s28 --threads=[P] -u[0,1] [m].
This runs algorithm A on the cross-product of the model m with the Büchi automaton of formula f. It uses a fixed hash table of size 2^{28} and P threads, and either subsumption (-u1) or not (-u0). The option ltl-semantics selects textbook LTL semantics as defined in [9, Ch. 4]. To investigate the overhead of CNDFS, we also run the multi-core algorithms for plain reachability on this crossproduct, even though this does not make sense from a model checking perspective. To compare effects of the search order on subsumption, we use both DFS and BFS.

Note finally, that we are only interested here in full verification, i.e. in LTL properties that are correct w.r.t the system under verification. This is the hardest case as the algorithm has to explore the full simulation graph. To test their on-the-fly nature, we also tried a few incorrect LTL formula for the above models, to which the algorithms all delivered counter examples within a second. But with parallelism this happens almost instantly [49, Sec. 4.2].

Implementation. LTSMIN defines a NEXT-STATE function as part of its PINS interface for language-independent symbolic/parallel model checking [24]. Previously, we extended PINS with subsumption [43]. `opaal` is used to parse the UPPAAL models and generate C code that implements PINS. The generated code uses the UPPAAL DBM library to implement the simulation graph semantics under *LU-extrapolated zones*. The LTL crossproduct [9] is calculated by LTSMIN.

LTSMIN's multi-core tool [68] stores states in one lockless hash/tree table in shared memory [69, 70]. For timed systems, this table is used to store *explicit state parts*, i.e. the locations and state variables [16]. The DBMs representing zones, here referred to as the *symbolic state parts*, are stored in a separate lockless hash table, while a lockless *multimap* structure efficiently stores full states, by linking multiple symbolic to a single explicit state part [43]. Global colour sets of CNDFS (*Blue* and *Red*) are encoded with extra bits in the multimap, while local colours are maintained in local tables to reduce contention to a minimum.

Hypothesis. CNDFS for untimed model checking scaled mostly linearly. In the timed automata setting, several parameters could change this picture. In the first place, the *computational intensity* increases, because the DBM operations use many calculations. In modern multi-core computers, this feature improves scalability, because it more closely matches the machine's

high frequency/bandwidth ratio [69]. On the other hand, the lock granularity increases since a single lock now governs multiple DBMs stored in the multimap [43, Sec. 6.1]. Nonetheless, for multi-core timed reachability, previous experiments showed almost linear scalability [43, Sec. 7], even when using other model checkers (UPPAAL) as a base line. On the other hand, the CNDFS algorithm requires more queries on the multimap structure to distinguish the different colour sets.

Subsumption probably improves the absolute performance of CNDFS. We expect that models with many clocks and constraints exhibit a better reduction than others. Moreover, it is known [12] that the reduction due to subsumption depends strongly on the exploration order: BFS typically results in better reductions than DFS, since "large" states are encountered later. CNDFS might share this disadvantage with DFS. However, as shown in [43], subsumption with random parallel DFS performs much better than sequential DFS, which could be beneficial for the scalability of CNDFS. So it is really hard to predict the relative performance and scalability of these algorithms, and the effects of subsumption.

Table 6.1: Runtimes (sec) and states counts *without* subsumption.

Model	P	$\|L\|$	$\|\mathcal{R}\|$	$\|V\|_{cndfs}$	$\|\Rightarrow\|_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	135449	438005	438005	1016428	26.1	26.2	27.8
csma	48	135449	438005	453658	1016428	1.0	0.9	0.9
fddi	1	119	179515	179515	314684	26.3	26.6	34.2
fddi	48	119	179515	566093	314684	1.6	0.7	2.7
fischer-1	1	521996	4987796	4987796	19481530	195.9	196.7	212.2
fischer-1	48	521996	4987796	5190490	19481530	4.8	4.6	5.1
fischer-2	1	358901	3345866	3345866	10426444	135.8	136.5	145.5
fischer-2	48	358901	3345866	3541373	10426444	3.4	3.3	3.7
train-gate	1	119989268	119989268	119989268	177201017	1608	1621	1724
train-gate	48	119989268	119989268	319766765	177201017	34.9	45.4	145.8

Experimental results without subsumption. We first compare the algorithms BFS, DFS (parallel reachability) and CNDFS (accepting cycles) without subsumption. Table 6.1 shows their sequential ($P = 1$) and parallel ($P = 48$) runtimes (T). Note that sequential CNDFS is just NDFS. We show the number of explicit state parts ($|L|$), full states ($|\mathcal{R}|$), transitions ($|\Rightarrow|$), and also the number of states visited in CNDFS ($|V|$). These numbers confirm the findings reported previously for CNDFS applied to untimed systems: The sequential run times ($P = 1$) are very similar, indicating little overhead in CNDFS. For the parallel runs ($P = 48$), however, the number of states visited by CNDFS ($|V|$) increases due to work duplication.

To further investigate the scalability of the timed CNDFS algorithm, we plot the speedups in Figure 6.7. Vertical bars represent the (mostly neg-

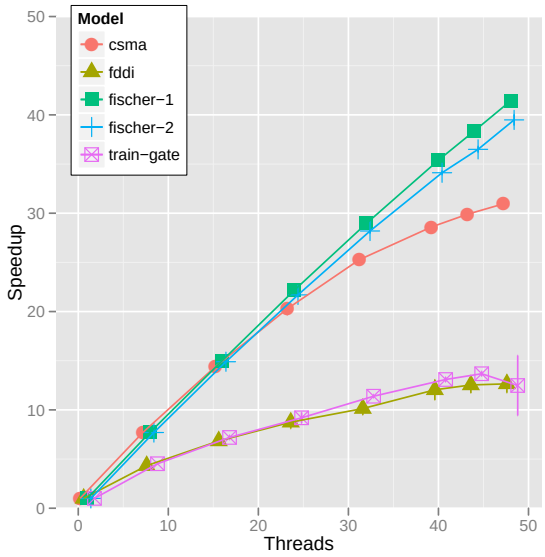


Figure 6.7: Speedups in LTSMIN/`opaal`

ligible) standard deviation over the five benchmarks. Three benchmarks exhibit linear scalability, while train-gate and fddi show a sub-linear, yet still positive, trend. For train-gate, we suspect that this is caused by the structure of the state space. Because fddi has only 119 explicit state parts, we attribute the poor scalability to lock contention, harming more with a growing number of workers.

Subsumption. Table 6.2 shows the experimental data for BFS, DFS and CNDFS with subsumption (Algorithm 12). The number of explicit state parts $|L|$ is stable, since reachability of locations is preserved under subsumption (Proposition 2). However, the achieved reduction of full states depends on the search order, so we now report $|\mathcal{R}|$ per algorithm, as a percentage of the original numbers.

We confirm [12] that subsumption works best for BFS reachability, with even more than 30-fold reduction for fddi, but none for train-gate (cf. column $|\mathcal{R}|_{bfs}$). For these benchmarks, the reduction is correlated to the ratio $X = |\mathcal{R}|/|L|$; e.g. $X \approx 1500$ for fddi and $X \approx 10$ for fischer. Subsumption is much less effective with sequential DFS, but parallel DFS improves it slightly (cf. column $|\mathcal{R}|_{dfs}$).

CNDFS benefits considerably from subsumption, but less so than BFS: we observe around 2-fold reduction for fddi, fischer and csma (cf. column $|\mathcal{R}|_{cndfs}$). Surprisingly, the reduction for parallel runs of CNDFS is not better than for sequential runs. One disadvantage of CNDFS compared to BFS is

Table 6.2: Runtimes and states counts *with* subsumption (in % relative to Table 6.1).

Model	P	$\lvert\mathcal{R}\rvert_{bfs}$	$\lvert\mathcal{R}\rvert_{dfs}$	$\lvert\mathcal{R}\rvert_{cndfs}$	$\lvert V\rvert_{cndfs}$	$\lvert\Rightarrow\rvert_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	48.7	88.9	58.3	94.7	41.2	41.3	90.3	95.2
csma	48	48.7	77.5	58.3	93.6	41.2	64.5	85.3	97.8
fddi	1	3.1	3.4	50.8	53.1	3.4	4.3	4.7	132.3
fddi	48	3.1	2.4	50.8	80.1	3.4	51.0	19.5	121.0
fischer-1	1	17.9	72.4	55.2	91.9	27.0	25.6	78.7	97.3
fischer-1	48	17.9	71.1	55.2	95.9	27.0	33.1	79.6	103.0
fischer-2	1	18.6	68.5	77.5	95.8	28.7	27.0	75.3	98.9
fischer-2	48	18.6	62.7	77.5	95.8	28.7	37.4	72.5	98.3
train-gate	1	100.0	100.0	100.0	100.0	100.0	100.6	100.6	104.3
train-gate	48	100.0	100.0	100.0	100.0	100.0	101.7	83.5	83.1

that only red states attribute to subsumption reduction. Probably some "large" states are never coloured red. We measured that for all benchmark models, 20%–50% of all reachable states are coloured red (except for fischer-2, which has no red states).

Subsumption decreases the running times for reachability: a lot for BFS, and still considerably for DFS, both in the sequential case and the parallel case, up to 48 workers. However, subsumption is less beneficial for the running time of CNDFS (it might even increase), but the speedup remains unaffected.

6.7 Viewed as Abstractions

In this section the view that the algorithms put forth in the paper can instead be reformulated as different abstractions over a lattice automaton will be explored. This will require that the classic NDFS algorithm be reformulated in terms of computing a fixpoint (albeit with a small trick to compute the fixpoint in correct DFS order), and subsequently this formulation can be extended to exploit subsumption. For the purpose of this exercise it is easier to work on the classic algorithm without any extensions for early termination – these extensions (the *cyan* color) only complicate the exercise, and are of no use when computing a fixpoint, because there is no possibility of early termination. Putting this into perspective the algorithms as formulated in Algorithm 10 and Algorithm 11 can be seen as *imperative* descriptions, while what is formulated in this section is a *declarative* description.

The classic Nested Depth First Search (NDFS) algorithm for a discrete state space ([93, 33]) can be reformulated as calculating a fixpoint in a lattice statespace of structure

$$\mathcal{S} \times (\mathcal{S} \cup \{\bot\}) \times \mathcal{L}_{color} \tag{6.1}$$

Algorithm 13 The classic NDFS algorithm [33], as formulated in [93, Fig. 1].

```
 1: procedure ndfs(s)
 2:     dfsBlue(s_0)
 3: procedure dfsBlue(s)
 4:     s.blue := true
 5:     for all t in NEXT-STATE(s) do
 6:         if ¬t.blue then
 7:             dfsBlue(t)
 8:     if s ∈ F then
 9:         seed := s
10:         dfsRed(s)
11: procedure dfsRed(s)
12:     s.red := true
13:     for all t in succ(s) do
14:         if ¬t.red then dfsRed(t)
15:         else if t = seed then
16:             report cycle
```

where $\mathcal{S}$ is the statespace, and $\mathcal{L}_{color}$ is an abstract domain given by Definition 46. The structure of the state space is, informally:

1. The current state of the automaton (blue/red search).
2. A seed accepting state, for which we are now searching for a path back to (red search); or $\bot$ if no accepting state has been seen (blue search).
3. The colors of the current state.

It should be noted that each state of this statespace closely correspond to a state of Algorithm 13.

Definition 46. *The NDFS color abstract domain is defined as* $\mathcal{L}_{color} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *where*

- D *is the set* $\{\bot, blue, red, \top\}$*, from which elements will typically be denoted* c*,*
- $\sqsubseteq$*,* $\sqcup$*, and* $\sqcap$ *is given by Figure 6.8.*
- $\bot$ *is an element representing no colors.*
- $\top$ *is an artificial greatest element, used to denote Büchi acceptance.*

The domain can be structured as such, due to the fact that a state that is *red* must already be *blue* [93], also formulated as (I0) on page 82 for the slightly modified algorithm.

The transition system for the state space from Equation 6.1 is given by lifting each transition $(s \to t)$ from the transition relation for $\mathcal{S}$ to $(s, seed, c) \to (t, seed', c')$ with the following transitions:

Figure 6.8: The NDFS color domain with the ordering shown as lines.

1. Add a non-deterministic transition if s is accepting and $seed = \bot$, with $seed' = s, c' = red$. This corresponds to the call to $dfsRed(s)$ on l.10 of Algorithm 13.

2. A normal transition:

   (a) if $t = seed$ then $c' = \top$, to capture the discovery of a Büchi accepting lasso, and $seed' = seed$. This corresponds to the check on l.15 of Algorithm 13.

   (b) otherwise: $seed' = seed, c' = c$, as the blue or red search continues. This corresponds to the recursive call on l.7 or l.14 of Algorithm 13.

Note that the order of these transitions is important, in order to preserve the required depth-first search order: transitions of type 2 needs to be explored depth-first, before transitions of type 1. The initial state is $(s_0, \bot, blue)$, where s_0 is the initial state of $\mathcal{S}$.

The ordering over the state space is given by

$$(s, seed, c) \sqsubseteq (t, seed', c') \iff s = t \wedge c \sqsubseteq c' \tag{6.2}$$

which captures that a state (regardless of *seed*) needs to be explored iff

(i) it wasn't explored before and should now be explored as *blue*

(ii) it was explored as *blue* but should now be explored as *red*

(iii) it was explored as *red* but now an accepting cycle was found, which needs to be propagated.

Theorem 2. *Computing the fixpoint of the transition system described using Algorithm 3 on page 31, with a worklist implemented as a stack, will result in a fixpoint* $f : \mathcal{S} \times (\mathcal{S} \cup \{\bot\}) \to \mathcal{L}_{color}$. *If* $\exists s \in S$ *s.t.* $f(s, s) = \top$ *then* $\mathcal{S}$ *has an accepting lasso.*

Furthermore, the worst-case time complexity of visiting each state at most twice in the case of no accepting lassos can be verified simply by looking at the ordering in Equation 6.3 and the structure of $\mathcal{L}_{color}$ in Figure 6.8: If no transition assigns the $\top$ value the fix-point value of any $s \in \mathcal{S}$ can increase at most twice (from $\bot$ to *blue*, and from *blue* to *red*). In case of an accepting lasso, the worst-case time complexity is visiting each state three times – a case avoided by the early termination of Algorithm 13.

For the case where the statespace $\mathcal{S}$ is actually a finite simulation graph $SG_{fin}(\mathbb{B})$ this construction of course works as well, due to Proposition 3 on page 79. Expanding the statespace structure using the definition of the simulation graph using Definition 41 on page 77 gives us:

$$\mathcal{S} \times (\mathcal{S} \cup \{\bot\}) \times \mathcal{L}_{color}$$
$$(L \times \mathcal{Z}) \times ((L \times \mathcal{Z}) \cup \{\bot\}) \times \mathcal{L}_{color}$$

where we recall that

L is the set of locations of the automaton

$\mathcal{Z}$ is the set of zones, typically represented as an element of the DBM domain, Definition 19 on page 24.

In this statespace we can define a different ordering, capturing the improvements put forth in Algorithm 11 on page 85:

$$((\ell, Z), seed, c) \sqsubseteq ((\ell', Z'), seed', c') \iff \ell = \ell' \wedge Z \sqsubseteq Z' \wedge \tag{6.3}$$
$$((Z = Z' \wedge c \sqsubseteq c') \vee \tag{6.4}$$
$$(red \sqsubseteq c' \wedge c \sqsubseteq c')) \tag{6.5}$$

where Equation 6.4 is the same condition as Equation 6.2, and Equation 6.5 is the condition that a state is subsumed if the subsuming state is at least *red*.

From this exercise a few interesting points can be extracted: it is immediately obvious that the ordering defined by Equation 6.3 is more fine-grained than the ordering of Equation 6.2. In addition, it shows how the NDFS algorithm can be implemented using a fixpoint algorithm such as Algorithm 3 on page 31, albeit with the small restriction of using a stack for the worklist to enforce a certain iteration order.

Chapter 7

An Automata-Based Approach to Trace Partitioned Abstract Interpretation

This chapter is based on the paper "An Automata-Based Approach to Trace Partitioned Abstract Interpretation" [83], currently under submission. Section 7.8 is an independent expansion for this thesis: it showcases some of the possibilities of having a model of the program as an artifact enables.

The paper proposes to use abstract interpretation to extract a lattice automaton (based on the control-flow graph of the program) with abstract transformers on the edges. This automaton, being an artifact in itself, can then be manipulated by the program analyst: edited, simulated, and verified. One possibility is to incorporate knowledge about the environment. Another is to do trace partitioning in order to gain precision.

It is shown how the model checker developed in Chapter 5 can be extended to incorporate joining of states, and how the resulting algorithm computes the exact same fixpoint as the classic worklist algorithm. The model checker benefits from using multiple cores.

Abstract

Trace partitioning is a technique for retaining precision in abstract interpretation, by partitioning all traces into a number of classes and computing an invariant for each class. In this work we present an automata-based approach to trace partitioning, by augmenting the finite automaton given by the control-flow graph with abstract transformers over a lattice. The result

is a lattice automaton, for which efficient model-checking tools exist. By adding additional predicates to the automaton, different classes of traces can be distinguised.

This shows a very practical connection between abstract interpretation and model checking: a formalism encompassing problems from both domains, and accompanying machinery that can be used to solve problems from both domains efficiently.

This practical connection has the advantage that improvements from one domain can very easily be transferred to the other. We exemplify this with the use of multi-core processors for a scalable computation. Furthermore, the use of a modelling formalism as intermediary format allows the program analyst to simulate, combine and alter models to perform ad-hoc experiments.

7.1 Introduction

The formal connections and similarities between model checking and static analysis are well known and have been explored by both Schmidt and Steffen [92], viewing static analysis as a model checking problem, and more recently by Nielson and Nielson [81], viewing model checking as a static analysis problem.

In this paper we exploit, and further explore, this deep connection between abstract interpretation and model checking by taking it in a more practical direction. In particular, we develop an automata-based approach to trace partitioning, as used in abstract interpretation, in which the control-flow graph of a program under analysis is transformed into a *lattice automaton* that can be model checked efficiently [42] to yield program analysis information. An overview of the proposed method can be seen in Figure 7.1. The proposed approach opens a very direct route for an implementation of

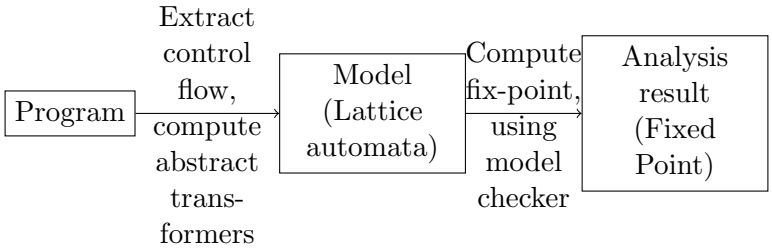


Figure 7.1: Overview of the proposed method.

an abstract interpretation to take advantage of highly-optimised state-of-the-art model checking engines and automatically gain by any performance improvements in the underlying model checking engine. In the following,

we illustrate this point by taking advantage of recent advances in model checking on multi-core platforms [43] to obtain an efficient implementation of abstract interpretation through trace partitioning.

In addition to efficient implementations, we believe that our approach also simplifies the often difficult task of using, adapting, and fine-tuning trace partitioning in abstract interpretation.

Finally, since programs in our approach are represented by expressive high-level models ready for model checking, it is easy to refine or add to the models. This is useful, e.g., for running simulations of the program; modelling the environment in which the program will run; or other systems, devices, or services that the program will interact with. These additional models can be as concrete or abstract as needed, e.g., using specifications to model services that have not yet been implemented.

As an example consider whether this program fragment is buggy:

```
1  char s[40];
2  gets();
3  printf("%s", s);
```

The answer should be: it depends. It could be the case that a guarantee can be made that any input will at most be 39 characters long. By extracting an abstract model of the program, this model can be edited by a program analyst to add details about the environment that are external to the program, in order to e.g. eliminate false warnings.

7.2 Related Work

The structure $A \times D$ where A is a finite set and D is a partially ordered but possibly infinite set, is a structure occuring:

- In model checking of infinite state systems (e.g. timed automata [43]) where the basis is a finite automaton with states A, enhanced with a symbolic part D where each element represents an infinite set of clock valuations.
- In static analysis using abstract interpretation [34], where the basis is a lattice D representing a program invariant over a set of traces, and A is used to divide the set of all traces into partitions (most commonly partitioning on the end-state of the trace) [76].

Additionally, a least upper-bound operator on D is used for doing approximate analysis: it is the basis of abstract interpretation [34], but also occurs in model checking of timed automata in the form of the convex hull abstraction [45].

Because the structure is so common, it is difficult to list all related work; we will focus on related work in the area of program analysis. We have

derived the proposed formalism from that used for model checking timed automata in UPPAAL [16], but it could as well be characterised as deriving from well-structured transition systems [51]. The addition of a join operator to the state space sets our definition apart from both, but the definition of abstraction operators has previously been done for e.g. timed automata [45] with the convex-hull abstraction for zones, or for finite lattices in finite automata in αSPIN [46]. In this work we address cases where the lattice has no infinite ascending chains, or where widening is applied to eliminate such chains.

Abstract interpretation [34] derives abstract transformers from the concrete semantics of a program, in relation to a given abstraction, namely a lattice. The abstraction is typically defined in terms of a Galois connection, or using widening/narrowing [37] where the latter is more powerful; in this work we focus on the Galois connection approach, and implicitly apply widening in place of joining to eliminate infinite ascending chains, as also done in [92]. Much work has been focused on finding abstract domains that are powerful enough to prove invariants of interest in real programs, while still being computationally affordable [40]. One technique for gaining precision without changing domain or sacrificing performance is *trace partitioning* [76, 61, 56, 90]. In this work we show how trace partitioning corresponds directly to instrumenting a finite automaton with additional predicates. [22] is another work in this area, describing a series of program analyses configurable in precision by a partial join operator, mimicking aspects of trace partitioning, but remarking that their base work-engine is a model checker.

The connection between abstract interpretation and model checking has been explored by Schmidt and Steffen [92], where it was shown how a parameterised variant of CTL-logic model checked over an abstract interpretation of a program can be used to answer data-flow queries. We build heavily on the work of [92], where the focus in [92] was on showing how model checking machinery could be used to solve the same problems as static analysis for models where a finite abstract model can be derived. Our focus is on showing that the *same* machinery can solve both model checking problems and static analysis problems, also for models where joining or widening is required for termination. In [38] it was shown how abstract interpretation can be used to reduce the state space needing to be searched by a model checker.

Model checking [9] was initially only applicable for small finite state automata. The state-space explosion problem meant that abstractions were needed [32] to reduce the state space to practical size. The case of finite domains [32, 4] allows the methods for finite state automata to be applied to these abstract models directly. In general, infinite domains are avoided because termination is not guaranteed.

In model checking software the counter-example guided abstraction refinement [59] approach allows model checking of increasingly more detailed abstractions of the program, starting from the control-flow graph. Based on found error traces additional boolean predicates are added to the lattice domain. CEGAR works very well in practice, but termination is not guaranteed.

Very efficient implementations of model checking algorithms for models with lattices exist, mostly in the area of timed automata model-checking [72, 43]. In particular model checkers such as LTSmin [68] exploit the multiple processing cores of modern shared-memory processors to do the work in parallel. The multi-core backend of LTSmin has recently been extended to the timed automata setting [43], with shown scalability up to 48 cores. The static analyser Astreé also has a parallel version [80], for which timings on a distributed cluster architecture (non-shared memory) are reported scaling up to 3–4 machines.

The work on lattice automata in [65] is unrelated to our definition of lattice automata, in that we allow arbitrary transitions as long as the transitions are monotonically enabled with regards to the lattice ordering, and [65] only allows transitions to affect the "value of a run" using the meet operator – which does not allow abstract transformers such as those of assignments.

7.3 Abstract Interpretation and Trace Partitioning

In this section we briefly review and define concepts and terminology related to trace partitioning in abstract interpretation, following Mauborgne and Rival [76].

A *program* is taken to be a transition system $(\mathcal{S}, \mathcal{A}, \rightarrow, s_0)$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions (statements), $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation, and s_0 is the initial state. Following convention, we write $s \xrightarrow{a} s'$ for the transition $(s, a, s') \in \rightarrow$.

A finite *trace* over a program is a finite sequence of states: $\sigma = s_0 \dots s_n$, such that for $0 \leq i \leq n$, $s_i \in \mathcal{S}$ and $s_i \xrightarrow{a_i} s_{i+1}$ for some $a_i \in \mathcal{A}$. We denote the final state of a trace σ as $\sigma_{\dashv}$. The set of all (finite) traces of a program P is denoted $[\![P]\!] = \{\sigma \in \mathcal{S}^* | \sigma \text{ is a finite trace of } P\}$ where $\mathcal{S}^*$ is the set of all sequences of states in $\mathcal{S}$.

In "standard" abstract interpretation, i.e., non-trace partitioning abstract interpretation, safety properties for a given program can be verified using approximations of the set of states that are reachable by the program. In the following we will approximate sets of traces, in anticipation of future developments.

The approximation of the set of traces are represented using an abstract domain, D, whose concomitant concretisation function maps an abstract

representation, $\ell \in D$, to the set of (program) traces represented by the abstract representation, i.e., $\gamma : D \to 2^{S^*}$. This gives the existence of a *Galois connection*, comprising α and γ s.t. $\alpha(X) \sqsubseteq \ell \iff X \sqsubseteq \gamma(\ell)$. This Galois connection, can be used to induce an abstract model [92] of a program $P = (\mathcal{S}, \mathcal{A}, \to, s_0)$, by defining for each concrete action a corresponding abstract action, $a \in \mathcal{A}$ $f_a : D \to D$, that safely approximates the concrete semantics by requiring that for all $s, s' \in S$ the following holds

$$s \xrightarrow{a} s' \text{ and } s_1 s_2 \cdots s \in X \text{ and } \alpha(X) = \ell \implies f_a(\ell) = \ell' \text{ and } s_1 s_2 \cdots s s' \in \gamma(\ell')$$

For any program, P, we denote this *abstract model* of (all the actions of) a program: $\mathcal{M}_P = \{f_a | a \in \mathcal{A}\}$.

In the above approach to abstract interpretation, all traces (corresponding to a given abstract value) are treated in the same way, since it is not possible to discern where the different traces originate from, e.g., whether or not a given trace corresponds to a 'then' branch of a conditional or to the 'else' branch. This may lead to an increased number of false positives when attempting to verify safety properties.

As shown in [76], it is possible to extract more information from the traces by *partitioning* the set of traces and thereby increase the precision of the analysis by treating each partition separately. Partitioning is performed through the use of a partitioning function:

Definition 47 (Partitioning function [76])**.** *A function $\delta : E \to 2^F$ is called a* partitioning function *if and only if it is* covering*:*

$$\bigcup_{e \in E} \delta(e) = F$$

and it is a partitioning *of F:*

$$\forall e, e' \in E \colon e \neq e' \implies \delta(e) \cap \delta(e') = \emptyset$$

In [76] it is proven that using trace partitioning leads to more precise analyses.

An example of a trace partitioning is the *final control state partition* that partitions traces based on their final state. This partitioning is commonly built into the abstract semantics in "standard" abstract interpretation. Following [76], we define a state to be a pair consisting of a control location, and a memory state: $\mathcal{S} = LOC \times MEM$. The final control state partitioning function is then $\delta_{LOC} : LOC \to 2^{\mathcal{S}^*}$, such that:

$$\delta_{LOC}(l) = \{\sigma \in S^* | \sigma_{\dashv} = (l, m) \text{ for some } m \in MEM\}$$

We can now define the result of a static analysis using abstract interpretation under a trace partitioning.

Definition 48 (Maximal Fixed Point Solution [82]). *Let $P = (\mathcal{S}, \mathcal{A}, \rightarrow, s_0)$ be a program, $\mathcal{M}_P$ an abstract model of P over the lattice $\mathcal{L}$, and $\delta : E \rightarrow 2^{\mathcal{S}^*}$ a partitioning function. The maximal fixed point solution (MFP) for the set of monotone framework equations for P is then a mapping $MFP_P : E \rightarrow D$, such that for any element $e \in E$: $MFP_P(e)$ is the least fixed point given $\mathcal{M}_P$ and P.*

The *MFP* is typically calculated using a worklist algorithm; for the final control state partitioning the instantiation is as shown in Algorithm 14.

Algorithm 14 The worklist algorithm for computing the MFP_P [82, p. 75], given abstract model $\mathcal{M}_P = \{f_a | a \in \mathcal{A}\}$, and initial lattice element ℓ_0. In the algorithm $s, t \in LOC$ and $a \in \mathcal{A}$.

```
procedure WORKLIST
  W := {(s,a,t) | (s,a,t) ∈ →}
  Analysis(·) := ⊥, Analysis(s_0) := ℓ_0
  while W ≠ ∅ do
    Remove some (s,a,t) from W
    if f_a(Analysis(s)) ⋢ Analysis(t) then
      Analysis(t) := Analysis(t) ⊔ f_a(Analysis(s))
      for all a', t' where t --a'--> t' do
        W := W ∪ {(t,a',t')}
  MFP := Analysis
```

More specialised partitioning functions can be defined, resulting in a more precise *MFP*. In [76] a number of partitioning functions are defined and discussed, partitioning on control flow and values, these will be treated in Section 7.5. A distinction should be made between static partitioning (where the trace partitioning is decided before the analysis, and does not change during the analysis) and dynamic partitioning (where the trace partitioning can change during the analysis). In Section 7.5 we will see how static partitioning allows for a more efficient encoding into an abstract model.

7.4 Lattice Automata

In the following we introduce the concept of a *lattice automata* and define model checking of lattice automata. Model checking is a well-known technique for verification purposes: it takes as input a model of the intended target system, e.g., a representation of a program, typically in the form of an automaton, and then computes the unfolded transition system of the automaton on-the-fly while checking all encountered states against the properties to be verified (usually formulated in a special logic). In this paper we will limit ourselves to *reachability properties*, namely whether a state with a certain property can be reached.

We start by defining *lattice transition systems*, a formalism that subsumes many other types of transition systems traditionally used in model checking, such as finite automata and timed automata.

Definition 49 (Lattice Transition System)**.** *A* lattice transition system *is a triple* $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ *where* S *is a finite set of states,* $\mathcal{L} = (D, \sqsubseteq, \sqcup)$ *is a lattice and* $\longrightarrow \subseteq S \times D \times S \times D$ *is a transition relation which has the* monotonicity property*: for all* $s_1, s_2 \in S$ *and* $\ell_1, \ell_2, \ell_1' \in D$ *if* $(s_1, \ell_1) \longrightarrow (s_2, \ell_2)$ *and* $\ell_1 \sqsubseteq \ell_1'$ *then* $(s_1, \ell_1') \longrightarrow (s_2, \ell_2')$ *for some* $\ell_2' \in D$ *with* $\ell_2 \sqsubseteq \ell_2'$*.*

Transitions are usually written as $(s, \ell) \longrightarrow (s', \ell')$ whenever $(s, \ell, s', \ell') \in \longrightarrow$. Configurations are pairs of the form (s, ℓ) where $s \in S$ and $\ell \in D$.

Definition 50 (Path)**.** *A finite path in a lattice transition system* $\mathcal{T}$ *is a finite sequence* $\sigma = (s_0, \ell_0)(s_1, \ell_1) \cdots (s_n, \ell_n)$ *such that* $(s_i, \ell_i) \longrightarrow (s_{i+1}, \ell_{i+1})$ *for all* i*,* $0 \leq i \leq n-1$*.*

We extend the $\sqsubseteq$ ordering to configurations such that $(s, \ell) \sqsubseteq (t, \ell') \iff s = t \wedge \ell \sqsubseteq \ell'$. Given a set of configurations X and a configuration (s, ℓ) we will write $(s, \ell) \sqsubseteq X$ to mean $\exists (s, \ell') \in X : \ell \sqsubseteq \ell'$.

To describe a lattice transition system in a concise way we will use *networks of extended lattice automata* (analogously to networks of timed automata as in UPPAAL [16]). An extended lattice automata is a finite automata extended with a finite set of integer variables of a finite domain. In UPPAAL, and our implementation in `opaal`, a restricted subset of the `C` programming language can be used to describe what conditions guard a transition, and how a transition updates the integer variables. For a network of n automata with states $\mathcal{S}_i$, and m integer variables over the finite domain $\{0, \ldots, max\}$ the set of states $\mathcal{S}$ of the network product is given by the crossproduct $\mathcal{S}_0 \times \cdots \times \mathcal{S}_n \times \{0, \ldots, max\}^m$, which is equivalent to a (large) finite automaton.

Denoting lattice elements by ℓ, ℓ' transitions can also be guarded by expressions over the lattice, e.g. $\ell \sqcap \ell' \neq \bot$, as long as the monotonicity property is satisfied. Note that the monotonicity property does not apply to guards or updates of the discrete variables. We will describe how a transition updates the lattice element from ℓ to ℓ' by an assignment of a expression using ℓ to ℓ', e.g. $\ell' = \ell \sqcap \ell''$. To describe an abstract transformation of an action a of the lattice element ℓ we will use the notation $\ell[\![a]\!]$, equivalent to applying the abstract transformer $f_a(\ell)$.

In a network of automata, different automata can synchronise over channels, such that one automaton initiates a synchronisation over channel `ch` using the syntax `ch!`, while another receives on the same channel: `ch?`. Synchronisations can either be one-to-one (handshake), or one-to-many (broadcast). Handshake synchronisation is blocking, and chooses a receiver nondeterministically among the enabled receivers. Unless otherwise noted all

synchronisations are handshake. An example of a network of extended lattice automata is shown in Figure 7.2.

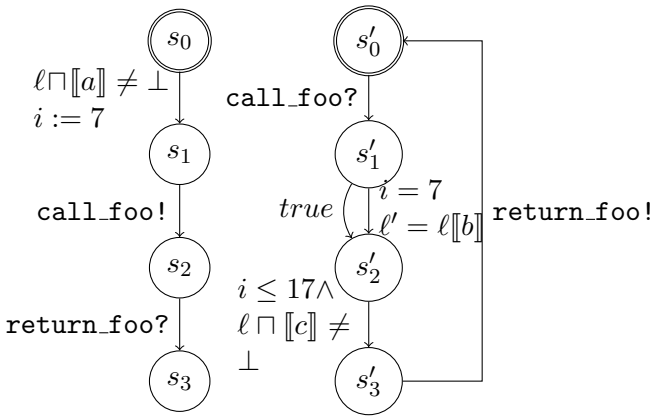


Figure 7.2: An example of a network of two lattice automata, with integer variable i, initial states s_0 and s'_0, and two channels `call_foo` and `return_foo`.

With the basic notions in place, we now turn to (reachability) model checking: model checking of lattice automata asks whether a model, $\mathcal{M}$, satisfies a formulae ϕ, expressed in some appropriate logic, written $\mathcal{M} \models \phi$. The result of solving a model checking problem instance is either a negative answer and a *counter-example* path, σ, or a positive answer and a set of configurations $\{(s_0, \ell_0), \ldots\}$ such that for any reachable state (s, ℓ) there exists some (s_i, ℓ_i) such that $(s, \ell) \sqsubseteq (s_i, \ell_i)$.

The requirement that a positive answer is accompanied by a set of configurations that cover all reachable configurations can be viewed as providing a certificate. It typically comprises the set of configurations examined during the model checking, the so-called "passed set" of all explored configurations. We call the set of configurations returned a *covering set*. Traditionally, the covering set is not presented to the user, due to the fact that its size may be exponential in the size of the input model. The covering set is related to the *coverability* problem of well-structured transition systems [54]. In the following we are only interested in using model checking to find a covering set, and thus assume the formula $\phi = true$, which is always satisfied.

We will now give two algorithms for solving the model checking problem for a lattice transition system.

Algorithm 15 is the algorithm typically used for model-checking reachability for timed automata, where the lattice $\mathcal{L}$ is the set of all zones (convex sets of clock valuations, efficiently representable as difference-bounded matrices), and $\sqsubseteq$ is the inclusion abstraction of [45]. If the set of reachable configurations in the model $\mathcal{M}$ is finite (typically because the lattice do-

Algorithm 15 Algorithm to compute a covering set or a counter-example, given a model in the form of a lattice transition system $\mathcal{M} = (S, \mathcal{L}, \rightarrow)$, initial configuration (s_0, ℓ_0) and formulae ϕ, if the reachable configurations in $\mathcal{M}$ is finite.

```
1:  procedure MC-COVER(M, (s_0, ℓ_0), φ)
2:    W := {(s_0, ℓ_0)}, P := ∅
3:    while W ≠ ∅ do
4:      Remove some (s, ℓ) from W
5:      if (s, ℓ) ⊭ φ then return counterexample
6:      if (s, ℓ) ⋢ P then
7:        for all (t, ℓ') s.t. (s, ℓ) → (t, ℓ') do
8:          W := W \ {(t, ℓ'')|ℓ'' ⊑ ℓ'} ∪ {(t, ℓ')}
9:        P := P \ {(s, ℓ')|ℓ' ⊑ ℓ} ∪ {(s, ℓ)}
10:   return Covering set P
```

main D is finite), Algorithm 15 will terminate.

Lemma 9. *If Algorithm 15 returns a covering set it is exact, i.e. some (s, ℓ) is covered by a reachable configuration if and only if $(s, \ell) \sqsubseteq P$.*

Sketch. For the if direction assume that some (s, ℓ) is covered by a reachable configuration. The algorithm will eventually visit some state (s, ℓ') with $\ell \sqsubseteq \ell'$ because of the monotonicity of $\rightarrow$, and add this to P, so eventually $(s, \ell) \sqsubseteq P$. To see that this holds invariantly afterwards notice that the only configurations removed from P in line 9, are covered by the newly added state thus preserving the invariant.

For the only if direction assume $(s, \ell) \sqsubseteq P$. Since the algorithm only adds a configuration (s, ℓ) to P if it is reachable and not already covered by P, the lemma holds. □

Algorithm 15 is only useful for finite state spaces, but provides exact answers. If a sound but over-approximated answer is sufficient, Algorithm 16 can be used. Algorithm 16 is the algorithm used for over-approximate reachability checking of timed automata using the convex-hull abstraction [45]. If the lattice $\mathcal{L}$ has no infinite ascending chains Algorithm 16 will terminate. If $\mathcal{L}$ has infinite ascending chains widening will have to be used instead of joining.

Lemma 10. *If Algorithm 16 returns a covering set it is sound, i.e. if some (s, ℓ) is covered by a reachable configuration then $(s, \ell) \sqsubseteq P$.*

Sketch. Assume (s, ℓ) is covered by a reachable configuration. Then at some point an (s, ℓ') with $\ell \sqsubseteq \ell'$ has been removed from W at line 4, because of the monotonicity of $\rightarrow$. At line 11 the invariant $(s, \ell') \sqsubseteq P$ (implying $(s, \ell) \sqsubseteq P$)

Algorithm 16 Algorithm to compute a covering set or a counter-example, given a model in the form of a lattice transition system $\mathcal{M} = (S, \mathcal{L}, \rightarrow)$, initial configuration (s_0, ℓ_0) and formulae ϕ, and using lattice join as abstraction.

```
1:  procedure MC-JOIN(M, (s_0, ℓ_0), φ)
2:    W := {(s_0, ℓ_0)}, P := ∅
3:    while W ≠ ∅ do
4:      Remove some (s, ℓ) from W
5:      if (s, ℓ) ⊭ φ then return counterexample
6:      if (s, ℓ) ⋤ P then
7:        for all (t, ℓ') s.t. (s, ℓ) → (t, ℓ') do
8:          ℓ_joined := ℓ' ⊔ ⨆{ℓ'''|(t, ℓ''') ∈ W ∪ P}
9:          W := W \ {(t, ℓ''')|ℓ''' ⊑ ℓ_joined} ∪ {(t, ℓ_joined)}
10:       ℓ'' := ℓ ⊔ ⨆{ℓ'''|(s, ℓ''') ∈ P}
11:       P := P \ {(s, ℓ')|ℓ' ⊑ ℓ''} ∪ {(s, ℓ'')}
12:   return Covering set P
```

will be established. Future modifications to P at line 11 preserves this invariant. □

Algorithm 15 was implemented in the multi-core backend of LTSmin with the purpose of model checking timed automata [43]. The performance and scalability of this algorithm was shown to scale almost linearly up to 48 processors, primarily limited by the size/structure of the model.

For the implementation of Algorithm 15 the disjunctive completion of the lattice $\mathcal{L}$ needs to be stored in general. In the implementation [43] this is done by storing states (s) in a shared passed-waiting hash table, and for each state storing a linked list of lattice elements $(\ell, \ell', \ldots)$ forming configurations $((s, \ell), (s, \ell'), \ldots)$, and a number of bits for each lattice element marking whether it is waiting or passed. This leads to scaling sub-linearly on models where there are many reachable configurations compared to the number of reachable states.

In this work we have extended the implementation to also have joining, providing a multi-core implementation of Algorithm 16. Because the implementation actually works on the disjunctive completion we can allow the join operator $\sqcup$ to be selective; it can select to keep two elements separate if so desired. This will be important for implementing dynamic partitioning. Note how Lemma 10 still holds in this case; on lines 9 and 11 only configurations actually covered by the joined lattice element are discarded.

7.5 Abstract Interpretation as Lattice Model Checking

In this section we describe how to concretely transform the problem of computing a *MFP* given a program $P = (\mathcal{S}, \mathcal{A}, \rightarrow, s_0)$ and abstract model of the program $\mathcal{M}_P$ over a domain D and a trace partitioning function δ, into a problem of computing a covering set for a lattice automaton. The presentation will be divided into four parts, depending on the nature of the trace partitioning function. Even though the most general case (dynamic partitioning) is sufficient, the simpler cases are crucial for the performance of the model checking.

7.5.1 Final Control Location Partitioning

The most abstract partitioning function we consider in this paper is the final control state partitioning δ_{LOC} as defined in Section 7.3. Recall that it partitions traces, based on the control location of the last state of the trace. Given a trace, it is thus sufficient to keep track of which control location the trace ends in, to know which trace partition the current memory state should be put in. Also recall that we assume program states are pairs in $LOC \times MEM$.

A finite automaton that accepts valid traces of the program P and at the same time keeps track of the current control location is the control-flow graph, given by the set of locations LOC and the set of edges $E \subseteq LOC \times LOC$ such that $(s, s') \in E$ iff $\exists a \in \mathcal{A}$ such that $s \xrightarrow{a} s'$. Consider the program in Figure 7.3a, for which the control-flow graph is shown in Figure 7.3b.

Given abstract semantics for the program P, $\mathcal{M}_P$ over some lattice $\mathcal{L} = (D, \sqsubseteq, \sqcup)$, we construct a lattice automaton $\mathcal{T} = (S, D, \Rightarrow)$ based on the control-flow graph:

S is the set of control locations LOC

D is the abstract domain as given by the abstraction

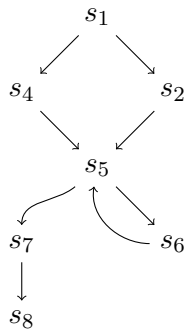
$\Rightarrow$ is the transition relation such that for a pair of configurations (s, ℓ) and (s', ℓ') it holds that $(s, \ell) \Rightarrow (s', \ell')$ if and only if $\exists a \in \mathcal{A}$ such that $s \xrightarrow{a} s'$ and $f_a(\ell) = \ell'$.

The lattice automaton for the program in Figure 7.3a is shown in Figure 7.3c, with the abstract transformers written on the edges as transformations of a lattice element ℓ into another ℓ'. Using Algorithm 16 a covering set can be computed for this lattice automaton.

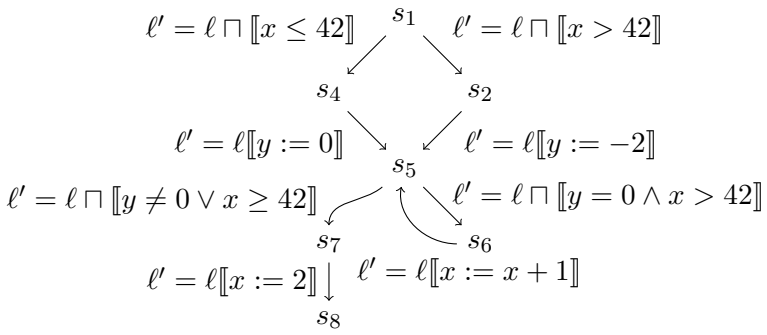
We can now state our main theorem:

```
1 if (x > 42)
2    y := −2
3 else
4    y := 0
5 while (y = 0 && x > 42)
6    x := x + 1
7 x := 2
8
```

(a) Program



(b) Control-flow graph



(c) Lattice automaton

Figure 7.3: (a) Program, (b) Control-flow graph of the program, and (c) constructed lattice automaton.

Theorem 3. *Given a program $P = (\mathcal{S}, \mathcal{A}, \rightarrow, s_0)$ and abstract semantics $\mathcal{M}_P$ over a domain D and the final control trace partitioning function δ_{LOC}, the MFP as computed by Algorithm 14 is the same as the covering set P computed by Algorithm 16 on the lattice automaton $\mathcal{T} = (S, D, \Rightarrow)$ constructed as described above.*

Sketch. For simplicity we assume the case of $\sqcup$ being a total function. The cardinality of the covering set, and *MFP* are then the same: one lattice element in LOC. We can therefore view P as a mapping $P : LOC \rightarrow D$, where $P(s) = \bot$ if there is no $(s, \ell) \notin P$. Similarly, we can consider the waiting list W as a function $W : LOC \rightarrow D$, because at any point in the algorithm there will be only one $(s, \ell) \in W$.

We show the proof in two parts: first we show that each iteration of Algorithm 16 can be simulated by a finite number of iterations in Algorithm 14. From [82, Sec. 2.4] we have that *Analysis* $\sqsubseteq$ *MFP* after each iteration of Algorithm 14. In the second part we show that at termination $P(s)$ is a fixed-point.

First part: each update of $P(s)$ in Algorithm 16 can be simulated by a finite number of updates of $\mathit{Analysis}(s)$ in Algorithm 14. Note that line 11 can be written as $P(s) := P(s) \sqcup \ell$, where (s, ℓ) was removed from W. Also, line 9 can be written as $W(s) := W(s) \sqcup P(s) \sqcup f_a(\ell)$ for some abstract transformer f_a.

We proceed by induction on the number of iterations in Algorithm 16. For the base case we have that the first update of P at line 11 must be of s_0, and because $P(s_0) = \bot$ we have that

$$P(s_0) := \bot \sqcup \ell_0 = \ell_0$$

This is simulated by the initial value of $\mathit{Analysis}(s_0)$ in Algorithm 14. In the first iteration, the configurations added to W in line 9 of Algorithm 16 are equivalent to adding the transitions to W in Algorithm 14.

In the inductive step we have that $P(s) := P(s) \sqcup \ell$ at line 11 must have produced ℓ as follows, on line 9 of some previous iteration:

$$\ell = W'(s) \sqcup P'(s) \sqcup f_b(\ell') \tag{7.1}$$

for some previous values of $W'(s)$ and $P'(s)$. By the induction hypothesis we have that $P'(s)$ is equal to $\mathit{Analysis}(s)$ for some iteration for some execution of Algorithm 14.

The value of $W'(s)$ is the result of the join of a number of lattice elements ℓ' found as successors in line 7, which is calculated as $\ell' = f_a(W'(s'))$ for some transition $(s', W'(s')) \Rightarrow (s, \ell')$. Thus the general form of $W'(s)$ is:

$$W'(s) = f_a(W'(s')) \sqcup W'(s) \sqcup P'(s)$$

for which $W'(s')$ can again be similarly decomposed as being calculated in some previous iteration. Because the number of iterations is finite, at some point it will be the case that $W'(s) = \bot$. Then we have that:

$$W'(s) = f_a(W'(s')) \sqcup \bot \sqcup P'(s) = f_a(W'(s')) \sqcup P'(s)$$

in which $f_a(W'(s'))$ can be similarly decomposed to a case where $f_a(W'(s')) = f_a(P''(s'))$, giving us

$$W'(s) = f_a(P''(s')) \sqcup P'(s)$$

which substituted back into equation (7.1) gives (because of the monotonicity of $P''(s) \sqsubseteq P'(s) \sqsubseteq P(s)$):

$$\begin{aligned}
\ell &= f_a(P''(s')) \sqcup P'(s) \sqcup P'(s) \sqcup f_b(\ell'') && (7.2)\\
P(s) &:= P(s) \sqcup f_a(P''(s')) \sqcup P'(s) \sqcup P'(s) \sqcup f_b(\ell'') && (7.3)\\
P(s) &:= P(s) \sqcup f_a(P''(s')) \sqcup f_b(\ell'') && (7.4)
\end{aligned}$$

This last equation is equivalent to two iterations of Algorithm 14 updating $P(s)$ given two different transitions, concluding the proof of this part.

Second part: At termination $P(s)$ is a fixed-point. Assume, towards a contradiction, that for some transition $s \xrightarrow{a} t$ it is the case that $f_a(P(s)) \not\sqsubseteq P(t)$. $P(s)$ was last updated in line 11, but before $P(s)$ was updated the abstract successor corresponding to the transition $s \xrightarrow{a} t$ was considered and a configuration $(t, f_a(\ell) \sqcup W'(t) \sqcup P'(t))$ was put on the waiting list W. Any update of $W(t)$ afterwards is monotonically increasing, until at some point later the configuration was removed from W, and either was already covered by $P(t)$ or $P(t) := P(t) \sqcup f_a(\ell) \ldots$ at line 11. Thus we have a contradiction.

Combining the two parts we have that after each iteration $P \sqsubseteq MFP$, and eventually P reaches a fixed-point: as MFP is the least fixed-point, $P = MFP$. □

7.5.2 Control Flow Based Partitioning

Another class of trace partitioning functions put forth is trace partitioning based on control flow [76, 56]. In general, control flow partitioning partitions traces based on their history of control flow choices, possibly merging the partitions at a later point in execution.

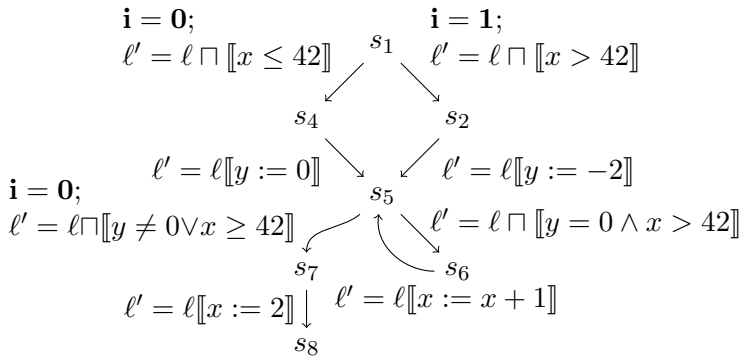
Lattice automata elegantly allow the recording of a limited amount of control flow history, by using discrete finitely valued integer variables. For each part control-flow partitioning point a discrete variable i is added, such that each branch of the control-flow point sets i to an unique value. If the partitions should later be merged [76] the variable is simply reset to one value. Consider the example lattice automata in Figure 7.4a, where traces are partitioned depending on the control flow at s_1, and merged at s_7.

Similarly loops can be unrolled any finite number of times by adding a loop counter variable that is reset on entry to the loop, and incremented on backedges until a certain limit. In fact, any iteration pattern that can be described by a finite automaton can be partitioned in this manner, e.g. partitioning the loop into whether the iteration count is even or odd: add variable i and annotate the backedge with $i = i + 1$ modulo 2.

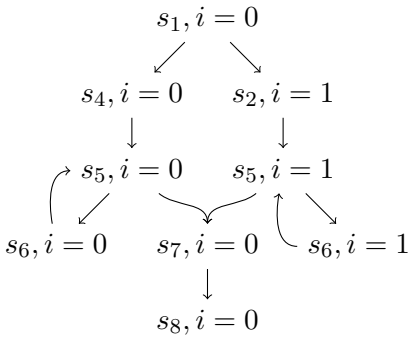
An advantage of using an intermediary format is that the program analyst can easily experiment with different control-flow partitionings by manually adding discrete variables and setting their value at different locations in a model editor. As long as no guards depend on the introduced variables, the soundness of properties is preserved.

7.5.3 Value Based Partitioning

Another class of trace partitioning is based on partitioning different values into different partitions. This can be handled similarly to the control-flow partitioning case, by splitting control flow into a finite set of value classes



(a) Lattice automaton with control flow partitioning, the only addition compared to Figure 7.3c being the updates of the variable i, as highlighted.



(b) Reachable locations for the lattice automaton in Figure 7.4a.

Figure 7.4: Control flow partitioning of the program in Figure 7.3a

(covering the entire range of the variable) using the general pattern shown in Figure 7.5. For partitioning into $v_0, \ldots, v_n$ different values a discrete variable i with range $[0, n]$ is added. At the partitioning point n transitions are added, each following the pattern in Figure 7.5. Each transition has a guard of the form $\ell \sqcap [\![x = v_1]\!] \neq \bot$ meaning that the transition can only be taken if at s the invariant $x = v_1$ is possible; there is no reason to explore a partition if it is already proven that no execution can have this value. If the transition is taken the partition is recorded in the discrete variable i, and the value v_i of the partition is assigned using the abstract transformer. If merging is desired at a later point, i is simply set to a constant value.

7.5.4 Dynamic Partitioning

The most general class of trace partitioning is allowing for the partitioning to be changed during computation [90]. In our setting this is realised using a *joining strategy* [42], namely allowing the $\sqcup$ function to be selective in which

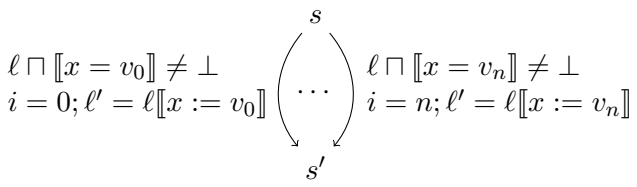


Figure 7.5: The general pattern of value based partitioning on a variable x into a finite number of partitions of values $v_0, \ldots, v_n$.

elements to join.

Definition 51 (Joining Strategy)**.** *A joining strategy is a function*

$$\delta : (S \times \mathcal{L}) \times (S \times \mathcal{L}) \to \{true, false\}$$

detailing whether two states in a lattice transition system are allowed to be joined, or should be kept separate.

A joining strategy can be used to define a partial join operator as

Definition 52 (Partial Join Operator)**.** *A joining strategy δ implies a partial join operator for a lattice transition system:*

$$\sqcup_\delta((s,\ell),(s',\ell')) = \begin{cases} (s, \ell \sqcup \ell') & \text{if } \delta((s,\ell),(s',\ell')) = true \\ (s,\ell) & \text{otherwise} \end{cases}$$

As mentioned in Section 7.4, Algorithm 16 is already designed for this.

During the analysis the joining strategy can be changed. One direction is to make the analysis coarser, based on the current analysis result or on extra-analysis information such as runtime and memory usage. A joining strategy δ_1 is (possibly) coarser than another δ_2 iff:

$$\forall s, \ell, s', \ell' : \delta_2((s,\ell),(s',\ell')) = true \implies \delta_1((s,\ell),(s',\ell')) = true$$

This is analogously to the ordering defined in [90], however it does suggest that the basis is a "completely partitioned system" and partitions are then merged to ensure termination.

A dynamically calculated joining strategy is however only limited by the answers it has already given and can be thought of as a sort of oracle. It can dynamically give answers that in turn create partitions, as long as no partitions overlap. This allows a joining strategy to exactly mimic the mechanisms put forth in [90]. It should be noted that static partitioning provides better performance than dynamic partitioning, because of the data structures used.

7.6 Experiments

To evaluate the feasibility and performance of the described approach, we have implemented a prototype for a small subset of `C`. The prototype is written in Python using the `pycparser` library, and generates models compatible with the `opaal` model checking framework [42]. One of the tools in `opaal` exports models to the multi-core model checker in the LTSmin toolset [68], previously developed for timed automata in [43]. The models can be edited in the UPPAAL [72] GUI, to introduce static partitionings.

We have implemented support for using the octagon domain [79] from the APRON library [62] (using the standard widening) in `opaal` models, and made the required changes to implement Algorithm 16 in the multi-core model-checker LTSmin; the change to the core algorithm implementation is 6 lines of code.

```
void main() {
    unsigned int a1, a2, a3,
          a4, a5, a6; int r;
    while (a1 < 20) {
        a1++; }
    while (a2 < 20) {
        a2++; }
    while (a3 < 20) {
        a3++; }
    while (a4 < 20) {
        a4++; }
    while (a5 < 20) {
        a5++; }
}
```

Figure 7.6: The `explode.c` program from [40]; with full control-flow trace partitioning there is an exponential number of traces to explore.

We have experimented with this prototype on a 8-core Intel Xeon X5570 machine, with 74Gb of RAM. In Figure 7.7 the runtimes for calculating a fix-point with increasingly more precise control-flow trace partitioning of the program in Figure 7.6, for which [40] reports that Astreé is unable to perform full trace partitioning. LTSmin has been invoked with the following command: `opaal2lts-mc --state=table -s 25 --threads=N -o bfs -u3 -prr` meaning to use a hashtable for passed-waiting list of size 2^{25}, run with N threads, use a breadth-first search order, do joining and chose a local successor state at random. Each experiment has been repeated 4 times and the mean is plotted, to account for the inherent non-determinism of the search order between multiple threads. As noted in [43] the search

order can have a large effect on the runtime, because one worker can find a "large" (according to the $\sqsubseteq$ ordering) state quickly, enabling another worker to skip part of the state space.

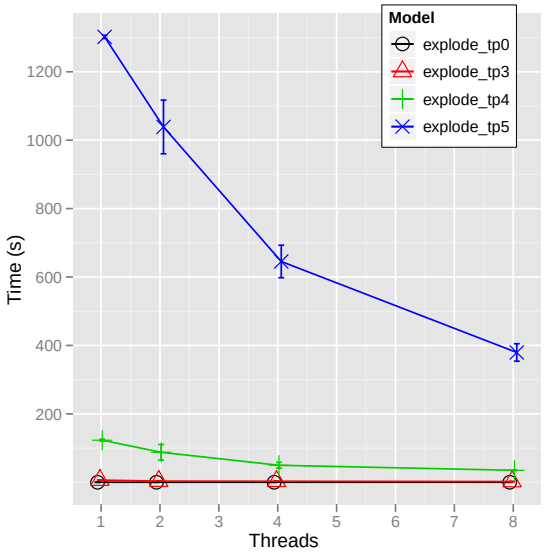


Figure 7.7: Benchmark timings (mean and standard deviation) for the `explode.c` program with full trace partitioning on the first 0, 3, 4 or 5 loops, run on 1, 2, 4 and 8 cores.

As can be seen in Figure 7.7 and Table 7.1 the use of more cores improves the runtime, in all cases except for no trace partitioning where the analysis time is so low that the thread initialisation is more expensive than the computation itself. Note that our runtimes cannot be compared to those in [40], as the domain in Astreé is more advanced than ours. The speedup is sub-linear, yielding speedups up to 3.5 using 8 cores. In [43], where the abstract domain was quite similar to the octagon domain namely Difference-Bound Matrices using UPPAAL's DBM library, the speedup was shown to be up to 40 using 48 cores. The difference in scalability can be attributed to two factors: implementation details affecting the multi-core performance[1] and the structure of the models allowing for less parallelism – as can be seen by the smaller speedup of models with little trace partitioning.

[1]E.g. increased usage of the dynamic memory allocator: APRON uses dynamic resizing of some data structures, whereas the UPPAAL DBM library does not. In general dynamic memory allocation is more expensive in a multi-core shared-memory setting, because it potentially requires synchronisation.

Table 7.1: Mean benchmark runtimes in seconds for the `explode.c` program with control-flow trace partitioning of the first 0, 3, 4 or 5 loops, run on 1, 2, 4 or 8 cores. (Relative speedups are in parentheses)

	Time_1	Time_2	Time_4	Time_8
explode_tp0	0.02s (1.00)	0.06s (0.39)	0.10s (0.25)	0.09s (0.26)
explode_tp3	6.34s (1.00)	3.41s (1.86)	3.01s (2.11)	2.18s (2.91)
explode_tp4	122.99s (1.00)	87.62s (1.40)	49.68s (2.48)	34.93s (3.52)
explode_tp5	1301.92s (1.00)	1038.70s (1.25)	645.40s (2.02)	379.44s (3.43)

7.7 Conclusion

We have shown the connection between abstract interpretation and model checking at a very practical level: by defining a formalism of lattice automata encompassing both domains, showing how this formalism can be used to compute a fix-point of an abstract semantics defined by a Galois connection and showing how trace partitioning is modelled very simply in this formalism. A common formalism as an intermediate format has the advantage that the intermediate format can be edited, debugged and simulated by a program analyst, to, e.g., add components modelling the environment external to the program.

A common formalism allows using the same machinery for solving problems from both domains. This approach has the advantage that improvements from one domain can immediately be transfered to the other, exemplified by using a multi-core model checker. This yields significant speedups, reducing analysis times by up to a factor 3.5 on a 8-core machine.

We plan to implement support for a much more complete subset of `C`, in order to perform more complete experiments. In addition we plan to implement support for more input programming languages; an especially exciting perspective is the ability to combine models extracted from different programming languages and model the interaction of e.g. `Python` code with `C`-code, or `C`-code with assembly.

Since the search order has a large impact on the speed-up it will be interesting to see how techniques from model checking, or from static analysis [55] can influence the performance – especially also in a multi-core setting, where different workers can employ different strategies.

Since we are using a model-checker we would like to implement support for asking the model-checker itself whether an error state or an `assert` is reachable. Since we are normally interested in all such possible violations, the logic formula would be a very large disjunction of predicates, which would need to be handled. It will also be interesting whether checking more elaborate properties, e.g. (bounded) liveness is possible.

Acknowledgments We would like to thank the LTSMIN and APRON developers for making their excellent code available to others in the research community.

7.8 Case Studies and Applications

In the previous sections it was shown how to use abstract interpretation to extract an lattice automata as an artifact. The machinery from Chapter 5 was extended to compute a program analysis result in the form of a fixpoint.

In this section a few simple case studies will be presented, of how this framework can be utilised to analyse programs in context. They fall into two categories: numerical analysis of C programs and worst-case execution time analysis of ARM binaries.

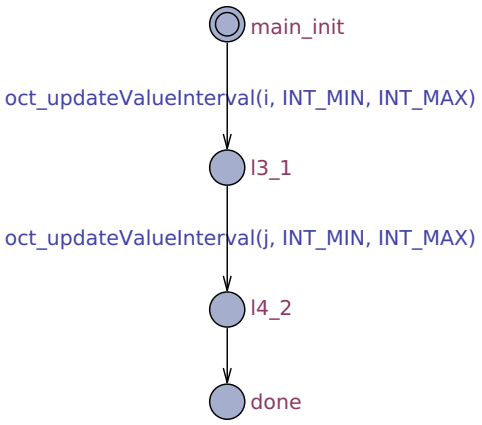
In Chapter 7 a Python prototype to extract models from C programs was introduced, C2OPAAL. In this section a number of case studies will be presented, showing the possibilities having such a model of the program enables.

7.8.1 Modelling the Environment

Consider the very simple program in Figure 7.8a. If executed and given a 0-character as input, it will crash with a "division by zero" exception.

```
1  int main() {
2    int i, j;
3    i = getchar();
4    j = 42 / i;
5    return j;
6  }
```

(a) The C source code.



(b) The lattice automaton generated by C2OPAAL when run on the program.

Figure 7.8: A program whose correctness depends on the environment in which it executes.

Using C2OPAAL the lattice automaton in Figure 7.8b is produced. Using the toolchain of Chapter 7 the octagon computed for l. 4 is:

$$\ldots i + 2147483648 >= 0; -i + 2147483647 >= 0 \ldots$$

meaning that $i \in [-2147483647, 2147483648]$ and that the division by i might divide by zero.

Indeed, this shows the prototype state of C2OPAAL; the call to `getchar` always returns something in the range $[-1, 255]$. Suppose this program is only called by another program, `caller`, and that `caller` writes exactly one upper-case alphabetical characters to this program. The program analyst can quite easily incorporate this information by editing the model, as shown in Figure 7.9. This produces a octagon for l. 4 that $i \in [65, 90]$, and thus no division by zero is possible.

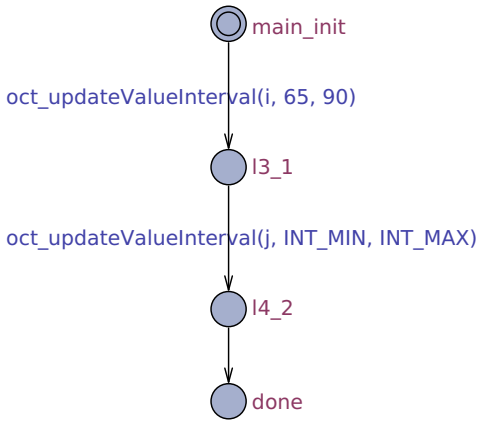


Figure 7.9: The edited lattice automaton of Figure 7.8b.

This example is admittedly very simplistic, and a similar effect might be obtained using e.g. source code annotations. The real advantage is that since the assumptions are part of the model itself any regular pattern of assumptions can be modelled, e.g. "first character is $[65, 90]$ then the next is $[32, 32]$, then $[65, 90]$, then $[32, 32]$ ...". The full power of the UPPAAL modelling language is available for use in the modification, e.g. adding additional synchronising automata, the subset of C that UPPAAL supports, etc.

7.8.2 Combining Models: Client-Server Communication

Instead of manually modelling the environment, it might be that the environment is instead given by another software component. Such is the case with e.g. applications communicating over a network or a hardware bus. A classic example of this is the client-server architecture, where a server

receives requests over the network from a client that in turn receives a response. The client and server might not be written in the same language, although for this example they will.

In UNIX-like systems communication over the network or between programs is abstracted through the use of *sockets*. After the setup of a network connection a socket is returned. To not get bogged down in details about how to setup a network connection, this example will use the standard input/output (the `stdin` and `stdout` sockets) as the communication medium.

```
1  int main() {
2    int ci, cj;
3    while ((ci = getchar()) != -1) {
4        if (ci >= 65 && ci <= 90)
5            putchar(ci);
6        else
7            putchar('␣');
8    }
9  }
```

(a) C source code for the "client" application. Accepts input, relays uppercase ASCII characters and replaces anything else by a space.

```
1  int main() {
2    int si, sj;
3    while ((si = getchar()) != -1) {
4        /* output 1 for A–Z input, 2 for space */
5        sj = 90 / si;
6        putchar(sj);
7    }
8  }
```

(b) C source code for the "server" application. Outputs 1 if given a uppercase ASCII character as input, and a 2 for a space.

Figure 7.10: A client-server application.

In Figure 7.10 a client-server application is given. The objective is to verify that the server application, when used by the client application, always outputs 1 or 2, i.e. in l. 6 of Figure 7.10b it holds that $sj \in [1, 2]$.

From each program a lattice automaton is extracted using C2OPAAL. These automata are then manually combined to a network of lattice automata, and are made to synchronise such that the client sending on l. 5 and l. 7 of Figure 7.10a synchronise to the server receiving on l. 3 of Figure 7.10b, thus emulating a blocking send/receive. If so desired the network

could also be modelled as an additional automaton in between, modelling the effects of buffers, network delay and packet loss. This process can of course be automated on a case-by-case basis, but for now is a manual process. The resulting model is displayed in Figure 7.11.

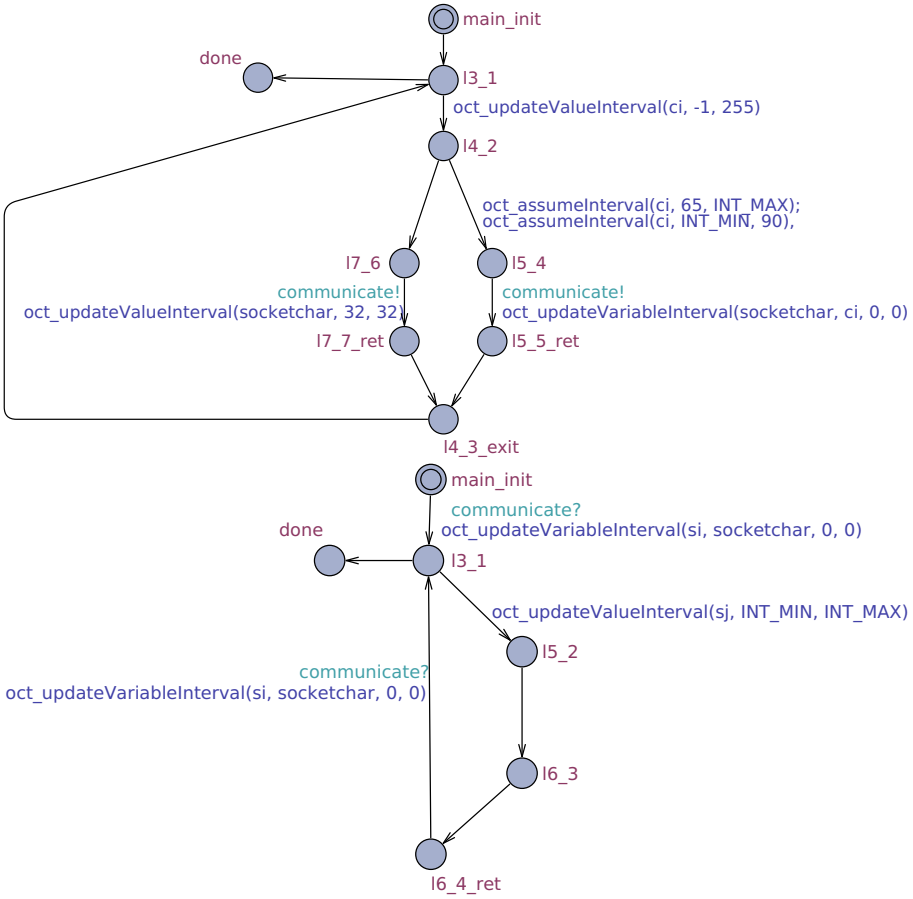


Figure 7.11: The manually composed client and server model of Figure 7.10. The two automata communicate over the octagon variable `socketchar` and synchronise over the channel `communicate`.

Some details have been edited that C2OPAAL does not yet support: the client call to `getchar` returns something in the range $[-1, 255]$, and the communication has been simplified by removing the various calls to `getchar` and `putchar`.

The division on l. 5 cannot be modelled exactly in the octagon domain [79], but instead it can be verified that on l. 5, $si \in [32, 90]$. The state space computed for the model in Figure 7.11 contains 46 states, of which the server process is in the location `l5_2` in 9 of them. Inspecting the octagon for each of these states reveal that either:

- `si - 32 >= 0 ; -si + 32 >= 0` i.e. $si \in [32, 32]$ or
- `si - 65 >= 0 ; -si + 90 >= 0` i.e. $si \in [65, 90]$.

Either joining these two intervals to $[32, 90]$ or keeping them separate and modelling the division in the interval domain confirms that

$$[90, 90]/[32, 90] = [1, 2]$$

thus allowing the verification that the client-server system always outputs either 1 or 2.

7.8.3 Combining Models: Embedding

In the previous example the client and server were both written in C. This need however not be the case, as long as a lattice automaton can be extracted from the program and the "glue" to combine models can be created, the programs need not be written in the same language. Another instance of this need to do program analysis across languages arises when analysing Python calling out to a C library, Python embedded in a C program, or a C program with embedded assembler. This aspect of cross-language program analysis will be explored by considering a small example of a C program with a small fragment of embedded ARM assembly, as given in Figure 7.12.

The C2OPAAL tool can generate a skeleton of the control-flow of the C code, but does not understand the ARM assembler. A separate prototype tool, ARM2OPAAL has been developed that extracts a lattice automaton, over the APRON octagon domain, for ARM binaries. After separating out the ARM assembly ARM2OPAAL can be used to generate a lattice automaton, that can subsequently be combined with the one extracted by C2OPAAL. At the moment this re-combination is a manual process, but it could be automated. The resulting model can be seen in Figure 7.13. The "glue" transitions assign the value of i to the register $r1$ when going from the C level to the assembly level, and assigns the value of $r0$ to j when going back.

In Figure 7.13 an assumption is added that the `getchar` call returns a lower-case alphabetical character, an ASCII value in $[97, 122]$. The model checker in turn returns a result that in l. 16 the value of j must be

$$j \in [65, 90]$$

meaning that the output is always an upper-case ASCII letter, as desired.

7.8.4 Worst-Case Execution Time Analysis of ARM Binaries

In [44] a framework for using model checking of timed automata to solve the worst-case execution time (WCET) analysis [99] for ARM binaries was presented. Following the work of Cassez et.al. [30] a program can be viewed

```
1   int main() {
2     int i, j;
3     i = getchar();
4
5     /*j = i - 'a' + 'A';*/
6     asm(
7         "sub r3, %1, #97"
8         "mov r4, #65"
9         "add r3, r3, r4"
10        "mov %0, r3"
11        : "=r" (j)
12        : "r" (i)
13        : "r3", "r4"
14        );
15
16    putchar(j);
17    return 0;
18  }
```

Figure 7.12: A C program, with embedded ARM assembly, having the same effect as the C fragment in the comment.

as a language generator, that the hardware in turn accepts at some speed and side-effect. That is, for a model P of the program, and a model H of the hardware, the model

$$P||H$$

models the execution of P on H.

Re-using the UPPAAL models of the hardware from [44, 30], and combining them with a network of lattice automata modelling the program extracted using e.g. ARM2OPAAL, gives such a combined model. The program will compute invariants using the APRON octagon domain, while the hardware will use the UPPAAL DBM domain to keep track of timing constraints, making the states of the system

$$(l_P, l_H, \ell_{octagon}, \ell_{DBM})$$

with partial ordering

$$(l_P, l_H, \ell_{octagon}, \ell_{DBM}) \sqsubseteq (l'_P, l'_H, \ell'_{octagon}, \ell'_{DBM})$$
$$\iff$$
$$l_P = l'_P \wedge l_H = l'_H \wedge \ell_{octagon} \sqsubseteq \ell'_{octagon} \wedge \ell_{DBM} \sqsubseteq \ell'_{DBM}$$

and a similarly defined joining operator. However, a joining strategy could be employed, to not get an overapproximation of the possible clock valuations, s.t.

$$\delta((l_P, l_H, \ell_{octagon}, \ell_{DBM}),(l'_P, l'_H, \ell'_{octagon}, \ell'_{DBM})) = true$$
$$\Longleftrightarrow$$
$$l_P = l'_P \wedge l_H = l'_H \wedge \ell_{DBM} \sqsubseteq \ell'_{DBM}$$

The benefits of such a combined model would be that any refinement of the program model to rule out infeasible paths, e.g. infeasible paths eliminated by the invariant tracking of the octagon domain or applying trace partitioning, will possibly improve the WCET estimate. A path leading to the returned WCET estimate can be returned, and the path can then be inspected using manual or automated techniques to ascertain its feasibility.

The implementation of the outlined approach is currently a work-in-progress, that will require some amount of practical work in maturing the ARM2OPAAL prototype, adapting the hardware models from [44] for usage with the `opaal` toolchain, and allowing the usage of multiple lattices in the same model.

One aspect that is essential for the sound application of any WCET method is the derivation of a sound overapproximation of the hardware, an aspect that will be explored in the next chapter.

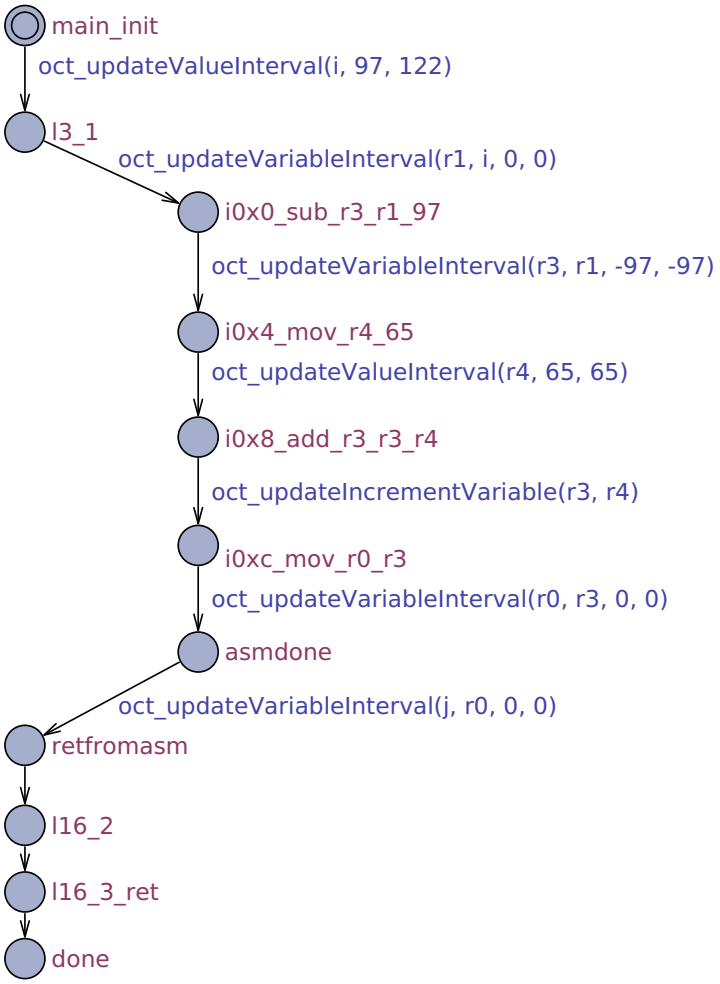


Figure 7.13: The manually combined lattice automaton for the program in Figure 7.12. The C level is on the left, while the ARM level is on the right, with "glue" transitions in between.

Chapter 8

What is a Timing Anomaly?

This chapter is based on the paper "What is a Timing Anomaly?" [31].

It is concerned with finding a good definition of *timing anomalies*, because the presence of these make finding good and sound abstractions for hardware very difficult. Ultimately, the goal would be to use the definition of timing anomalies in the worst-case execution time analysis of programs, by e.g. defining a lattice-based abstraction of the hardware, such as done for caches in [3].

Abstract

Timing anomalies make worst-case execution time analysis much harder, because the analysis will have to consider all local choices. It has been widely recognised that certain hardware features are timing anomalous, while others are not. However, defining formally what a timing anomaly is, has been difficult.

We examine previous definitions of timing anomalies, and identify examples where they do not align with common observations. We then provide a definition for *consistently slower hardware traces* that can be used to define timing anomalies and aligns with common observations.

8.1 Introduction

Developing reliable real-time systems requires that guarantees on the runtime of tasks can be given, that hold under all circumstances i.e. regardless of the input data and previous execution history of the system. Typically the Worst-Case Execution Time (WCET) is the most important guarantee as it can be used to ensure the system responds in a timely manner.

However, modern processors are not optimized for worst cases, but optimize for improving the average case performance instead. This often makes

their worst-case behaviour much harder to predict, and thus makes it harder to give absolute guarantees. One often hoped for property is that local worst-case timing choices will lead to the global worst-case timing — when this is not the case it is dubbed a *timing anomaly*. The classic example of a timing anomaly [75] is shown in Figure 8.1, where a cache miss for instruction A (bottom) is locally slower but turns out not to be the globally slowest (the top trace is slower). The example will be treated in greater detail later.

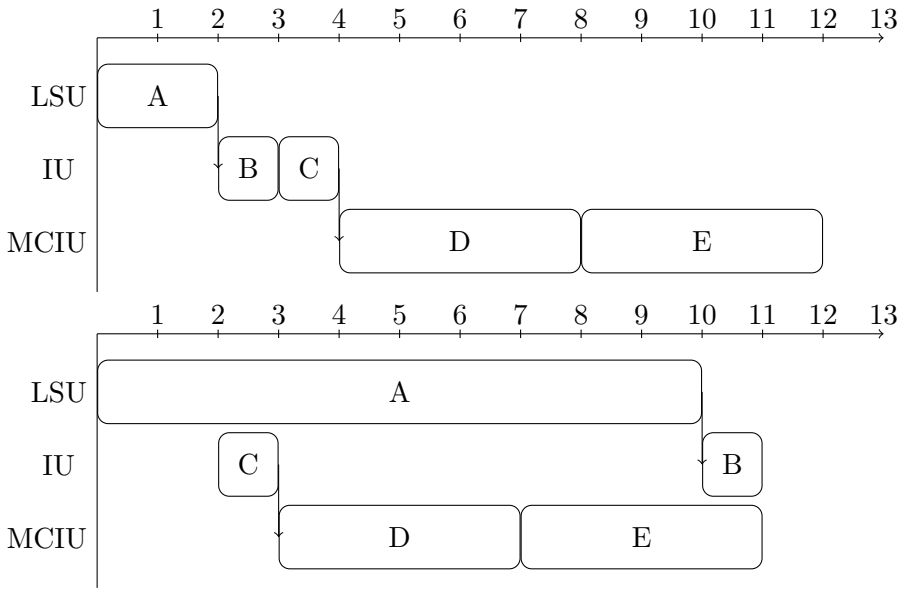


Figure 8.1: The canonical example of a timing anomaly from [75], where a cache miss (locally slower) leads to a scheduling that is globally faster. LSU, IU and MCIU are the three functional units that can execute out-of-order, but preference is given to older instructions.

If an execution platform can be proven to be free of timing anomalies, very efficient techniques exist for analysing the worst-case timing behaviour [99]. On the contrary, if the execution platform exhibits timing anomalies there is little hope for using the same efficient abstraction techniques [75].

Because of this, identifying timing anomalies has been an area of interest for some time, and some observations have been broadly recognised as being true:

- The LRU cache replacement policy is not timing anomalous.
- Other cache replacement policies such as FIFO and MRU exhibit timing anomalies [21, 53].
- In-order pipelines (without caches) are not timing anomalous.

- Resource allocation decisions (such as those presented by out-of-order execution or cache replacement) are a necessary condition for timing anomalies [98].

Using efficient abstraction techniques to compute the WCET is at the core of WCET analysis tools. However, the most powerful abstractions are sound only for timing anomaly free hardware. This explains why there have been some attempts to formally define timing anomalies [75, 89], but the various definitions have not been related to each other thus far.

In this work we will argue that the previous attempts are either too coarse or too precise to be used as universal definitions of timing anomalies. Each of the previous attempts definitely have their merits for application in connection with different analysis techniques (abstract interpretation, etc.), but a definition of timing anomalies should be as general as possible, while still retaining the property that the existence of timing anomalies forces the WCET analysis to consider more than one local choice.

Our Contribution

Our work is guided by the need for a definition of timing anomalies on the concrete model of the processor, instead of abstractions thereof. Consequently, in the following we propose a definition of timing anomalies that can be used in two different directions:

1. on hardware systems that are proven to be free of timing anomalies, the efficient abstraction techniques used in most WCET analysis tools are sound;

2. the definition we propose is based on the concrete reference hardware and only relates comparable hardware traces in order to avoid spurious timing anomalous diagnostics (see Section 8.4.2) resulting from abstraction of the hardware and/or of the hardware traces.

Without a definition of timing anomalies on the concrete reference hardware model, it is impossible to prove that abstraction is sound. Therefore we define timing anomalies as a property over different traces of the concrete hardware model.

But what traces should be comparable? We will argue that only traces resulting in the same instruction stream, i.e. the same program execution, should be comparable, in particular traces produced by different input data should not be comparable. It seems natural that different input data can result in different control flows, and therefore different instruction streams, where small changes in the input can result in much longer instruction streams, and therefore much longer execution times.

Another consideration is what elements of the hardware traces should be compared. Previous definitions have compared the timing of the first

instruction with the timing of the last instruction in the stream [75], or made comparisons at points where the two traces have executed the same number of instructions [48]. We will argue that comparisons should be made on the *completion times* of each instruction.

Outline of the Paper

This work is divided into five sections: In Section 8.2 we define hardware systems and execution of programs on them. In Section 8.3 we define timing anomalies, and then examine related work in Section 8.4. We then compare the different definitions in Section 8.5, before concluding in Section 8.6.

8.2 Execution of Programs on Hardware

Before turning to timing anomalies, we first need to formalise our notion of hardware systems and how programs are executed on them. In order for our work to be applicable to a wide variety of systems, we aim to make as few assumptions about the hardware as possible. However, it usually consists of a processor and main memory (including caches). As usual, the hardware can process (machine code) instructions, taken from the set $\mathsf{Instructions}$, with each instruction located in memory at some address. We let $\mathsf{HardwareStates}$ be the (finite) set of possible hardware states and assume the hardware states contain the state of the memory.

The *semantics* of a hardware system is given by a transition system that specifies how the state of the hardware evolves in order to execute a program on given input data. We only model transitions between hardware states that take an observable amount of time and produce an observable result[1], e.g., finishing execution of a (set of) instruction(s).

The observable results, in the set $\mathsf{Observations}$, are not strictly necessary but are admitted as a convenience for later developments. In our work, the typical observations of interest in a given hardware system are the instructions that finish (in each cycle or time unit). We can now give the formal definition of a hardware system.

Definition 53 (Hardware System)**.** *A* hardware system $\mathcal{H}$ *is formalised as a* stutter-free *and* deterministic *labelled transition system*

$$\mathcal{H} = (\mathsf{HardwareStates}, \mathsf{Time} \times \mathsf{Observations}, \rightarrow).$$

The transition relation

$$\rightarrow \subseteq \mathsf{HardwareStates} \times (\mathsf{Time} \times \mathsf{Observations}) \times \mathsf{HardwareStates}$$

[1] Bus latency, speculative execution, pipeline flushes, etc., are not visible and may generate extra cycles before an externally visible hardware state occurs.

describes the time *required to reach the next state and the externally visible* observations *produced by a transition.*

As usual, a transition $(s, (t, o), s') \in \rightarrow$ is denoted $s \xrightarrow{(t,o)} s'$. The properties "stutter-free" and "deterministic" can then be formulated as follows: if $s \xrightarrow{(t,o)} s'$ then $s \neq s'$, and if $s \xrightarrow{(t,o)} s'$ and $s \xrightarrow{(t',o')} s''$ then $t = t', o = o'$ and $s' = s''$. A *run* in the hardware system $\mathcal{H}$ is defined to be a sequence $\sigma = s_0 \xrightarrow{(t_1,o_1)} s_1 \xrightarrow{(t_2,o_2)} \cdots \xrightarrow{(t_n,o_n)} s_n$ such that for all $1 \leq i \leq n-1$ it holds that $s_i \xrightarrow{(t_{i+1},o_{i+1})} s_{i+1}$ (in the $\mathcal{H}$ transition system) and the *length* of the run is defined as $\mathit{length}(\sigma) = n$. The *trace* of the run σ is $\mathit{trace}(\sigma) = (t_1, o_1) : (t_2, o_2) : \cdots : (t_n, o_n)$; the *time trace* of σ is $\mathit{time}(\sigma) = t_1 : \cdots : t_n$, and the *observation trace* of σ is $\mathit{obs}(\sigma) = o_1 : o_2 : \cdots : o_n$.

Example 1. *Observing the two traces shown in Figure 8.1 using "just finished instructions" as observations, we obtain the following run for the first (top) part of the example:*

$$h_0 \xrightarrow{(2,\{A\})} h_1 \xrightarrow{(1,\{B\})} h_2 \xrightarrow{(1,\{C\})} h_3 \xrightarrow{(4,\{D\})} h_4 \xrightarrow{(4,\{E\})} h_5$$

and the run below for the second (lower) example in Figure 8.1:

$$h_0 \xrightarrow{(3,\{C\})} h_1 \xrightarrow{(4,\{D\})} h_2 \xrightarrow{(3,\{A\})} h_3 \xrightarrow{(1,\{B,E\})} h_4$$

Note that the observation on the final transition above shows that the two instructions labelled B and E finish simultaneously.

8.2.1 Execution of a Program on Hardware

We assume that all programs terminate. Given a program P and input data d in $\mathsf{Data}(P)$ (the set of admissible input data for P), the language semantics uniquely determine the *program trace*, i.e. the sequence of instructions to be performed to compute the result of program P on input d. In the following we need to be able to unambiguously identify specific occurrences of instructions in a program trace (the same instruction can be performed several times in the trace, for instance when loops are executed). Thus we formalise the program trace for (P, d) to be a mapping that assigns a unique index to each instruction in the program trace: $\mathsf{ProgramTrace}(P, d) : [1..k] \rightarrow \mathsf{Instructions}$ where k is the length of the trace and $\mathsf{ProgramTrace}(P, d)(j)$ gives the instruction executed at step j for each index $1 \leq j \leq k$.

Example 2. *The program trace for the example in Figure 8.1 is:*

$$\mathsf{ProgramTrace}(P, d) = [1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E]$$

Given a hardware system $\mathcal{H}$, a program P and input data $d \in \mathsf{Data}(P)$, we let $I(P,d) \subseteq \mathsf{HardwareStates}$ be the hardware states that contain program P and input data d in memory, and where the first instruction of P is about to start execution. For $h_0 \in I(P,d)$, executing P with input d on $\mathcal{H}$ yields a unique sequence[2] of transitions in the hardware: $h_0 \xrightarrow{(t_1,o_1)} h_1 \cdots h_{n-1} \xrightarrow{(t_n,o_n)} h_n$. As the hardware is deterministic, each observation o_i can be taken to be a set of indices in $\{1,\ldots,k\}$: the indices uniquely identify the occurrences of instructions of $\mathsf{ProgramTrace}(P,d)$ being completed at each step. We can now formalise what it means to execute a program on a hardware system:

Definition 54 ((P,d,h_0)-run)**.** *Let $\mathcal{H}$ be a hardware system, P a program, d input data, and $h_0 \in \mathsf{HardwareStates}$. Then the run (in $\mathcal{H}$) $h_0 \xrightarrow{(t_1,o_1)} h_1 \cdots h_{n-1} \xrightarrow{(t_n,o_n)} h_n$ is called a (P,d,h_0)-run (in $\mathcal{H}$) whenever $h_0 \in I(P,d)$ and o_i is the set of (indices of) instructions completed during the transition $h_{i-1} \xrightarrow{(t_i,o_i)} h_i$ for $1 \leq i \leq n$.*

Definition 55 (Completion Time)**.** *Let*

$$\mathsf{ProgramTrace}(P,d) : [1..k] \to \mathsf{Instructions}$$

be the program trace of (P,d) and let σ be the corresponding (P,d,h)-run starting in $h \in I(P,d)$ with trace$(\sigma) = (t_1,o_1) : \cdots : (t_n,o_n)$. By $Ctime(\mathsf{ProgramTrace}(P,d)[j],h)$ we denote the completion time *of instruction $1 \leq j \leq k$ finishing after transition m (i.e., $j \in o_m$) and define it as follows $Ctime(\mathsf{ProgramTrace}(P,d)[j],h) = \sum_{i=1}^{m} t_i$.*

We let

$$Ctime(\mathsf{ProgramTrace}(P,d),h) = \max_{1 \leq j \leq k} Ctime(\mathsf{ProgramTrace}(P,d)[j],h)$$

denote the maximal completion time for all instructions in the program and thus for completing the entire program.

Example 3. *The first trace in the example in Figure 8.1 has the following completion times for the 5 instructions: $[2,3,4,8,12]$ and for the second trace: $[10,11,3,7,11]$.*

8.2.2 Exemplary Hardware Models

To be able to exemplify different phenomena we will use three different hardware models:

[2]Which we assume to be correct with regard to the instruction semantics.

M_1 is a single-stage pipeline with a data-cache. The instructions of interest are the memory accesses, and the hardware model will be used to demonstrate timing anomalies with different cache replacement policies such as LRU and FIFO. For this reason we will simply denote instructions by the memory address they access.

M_2 is the simplified PowerPC architecture described in [75]. It is an out-of-order processor with three functional units: a Load/Store unit (LSU) communicating with a data cache, a Multi-Cycle Integer Unit (MCIU) and an Integer Unit (IU). For a detailed description see [75]. It is used for the classic example in Figure 8.1.

M_3 is a single-stage pipeline with no caches, but with a multiplication instruction `MUL` that takes 1 cycle if one of the operands is 0, and 2 cycles otherwise. This is a simplified version of the processor model in [48]. We extend M_3 with conditional execution of all instructions as on the ARM architecture [7].

8.3 Formalising Timing Anomalies

Slightly simplified, our notion of timing anomaly is based on the idea that timing anomalies only occur when a program is executed on a hardware system where no initial state gives rise to worse (slower) execution time than all other initial hardware states (modulo "irrelevant" parts of the hardware state). This approach requires us to formalise what it means for one program execution to be slower than, or rather: consistently as slow as, another execution (of the same program on the same data):

Definition 56 (Consistently as slow)**.** *Let P be a program with input data d and let* $\mathsf{ProgramTrace}(P,d) : [1..k] \to \mathsf{Instructions}$ *be the program trace of (P,d). Let $h, h' \in I(P,d)$ and σ (respectively σ') be a (P,d,h)-run (respectively (P,d,h')-run). Then h' is said to be* consistently as slow *as h, denoted $h \sqsubseteq_{time} h'$, if and only if*

$$\forall 1 \leq j \leq k\colon Ctime(\mathsf{ProgramTrace}(P,d)[j], h) \leq Ctime(\mathsf{ProgramTrace}(P,d)[j], h')$$

Intuitively the above definition compares the execution time of all prefixes of a program trace and requires one to be consistently as slow as the other.

Example 4. *Consider the example in Figure 8.1, where h is the hardware state resulting in the top run, and h' the state resulting in the bottom run.*

$$Ctime(\mathsf{ProgramTrace}(P,d)[1], h) = 2 \leq Ctime(\mathsf{ProgramTrace}(P,d)[1], h') = 10$$

meaning instruction A was slower in the bottom trace, but

$$Ctime(\mathsf{ProgramTrace}(P,d)[5], h) = 12 \not\leq Ctime(\mathsf{ProgramTrace}(P,d)[5], h') = 11$$

meaning instruction E was not slower in the top trace, thus $h \not\sqsubseteq_{time} h'$. However instruction A in the bottom trace is still slower than in the top trace, so $h' \not\sqsubseteq_{time} h$.

The "consistently as slow" ordering is a pre-order (see below). However, it is not a partial order since two hardware states, which differ only in parts that are irrelevant to a given program, will still give rise to identical instruction completion times:

Lemma 11. *For all programs P and input data d the relation $\sqsubseteq_{time}$ is a pre-order on $I(P, d)$.*

We can now propose a formal definition for timing anomalies:

Definition 57 (Timing Anomaly Free)**.** *A hardware system, $\mathcal{H}$, is said to be* free of timing anomalies *with respect to program P and input d, if and only if there exists a maximal element, $\mathcal{W} \in I(P, d)$: $\forall h \in I(P, d)$: $h \sqsubseteq_{time} \mathcal{W}$.*

Note that the maximal element is not necessarily unique: consider the case of LRU caches with no useful elements in them, hence all references resulting in cache misses.

The following lemma characterises (the absence of) timing anomalies in terms of upper bounds for arbitrary pairs of states. As shown in Section 8.5, this characterisation can be convenient when proving the presence of timing anomalies.

Lemma 12. *A hardware system $\mathcal{H}$ is free of timing anomalies with respect to program P and input data d if and only if $\forall h, h' \in I(P, d)$: $\exists h'' \in I(P, d)$: $h \sqsubseteq_{time} h'' \wedge h' \sqsubseteq_{time} h''$.*

Proof. The "only if" direction is trivial and the "if" direction is proved by induction in the size of $I(P, d)$. □

Note that this does not require all hardware states to be ordered under $\sqsubseteq_{time}$, but only requires that for any pair of states a third state exists that gives rise to a consistently as slow run as both; i.e. it should be an upper bound for the pair of states.

Consider the example in Figure 8.2 where two runs of a LRU cache are not ordered either way, but we would still like to characterise LRU as not timing anomalous; there exists a consistently slower initial state than both, namely the empty cache.

Having defined timing anomaly free-ness for a given program and input data, it is straightforward to generalise the definition to cover entire hardware systems:

Definition 58 (Timing Anomaly Free Hardware System)**.** *A hardware system, $\mathcal{H}$, is said to be* free of timing anomalies *for program P if and only if it*

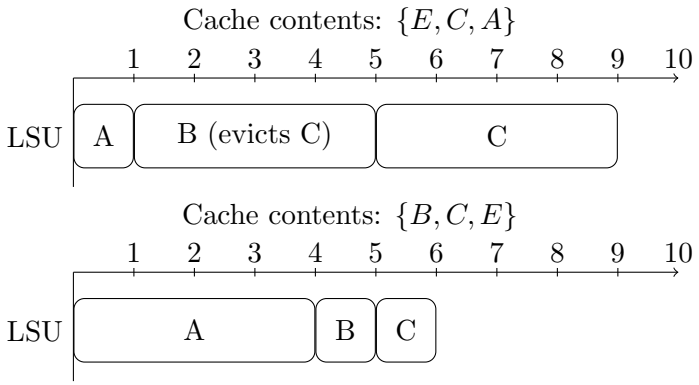


Figure 8.2: Two runs of the program `LD A; LD B; LD C` on hardware model M_1 with a LRU cache. The cache contents are ordered sets of data elements, from newest to oldest.

is timing anomaly free for each $d \in \mathsf{Data}(P)$*. Hardware* $\mathcal{H}$ *if free of timing anomalies if and only if it is timing anomaly free for all programs* P *(valid for* $\mathcal{H}$*).*

Finally we relate our definition of "consistently as slow as" to the definition of the WCET for a program P on hardware $\mathcal{H}$.

Definition 59 (Worst Case Execution Time (WCET))**.** *The* worst case execution time *for a program* P *(on* $\mathcal{H}$*) is defined as follows:*

$$WCET_{\mathcal{H}}(P) = \max_{h \in I(P,d), d \in \mathsf{Data}(P)} \{Ctime(\mathsf{ProgramTrace}(P,d), h)\}$$

If $\mathcal{H}$ is free of timing anomalies for P, only a maximal element in $I(P,d)$ need be considered. Indeed, if $h \sqsubseteq_{time} h'$, then $Ctime(\mathsf{ProgramTrace}(P,d), h) \leq Ctime(\mathsf{ProgramTrace}(P,d), h')$. Definition 59 is then reduced to computing $\max_{d \in \mathsf{Data}(P)}\{Ctime(\mathsf{ProgramTrace}(P,d), h) | h \text{ maximal in } I(P,d)\}$.

8.4 Related Work

8.4.1 Defining Timing Anomalies by Changes in Instruction Latency

Timing anomalies were first discovered by Lundqvist and Stenström in [75, 74]. Their definition is re-used in [98], and we formulate it here in our framework.

Assume a sequence of instructions $\pi = i_1 : \cdots : i_n$, with corresponding latencies $\tau_\pi(i_j)$, and total execution time C. Consider a situation where there exists a latency variation, Δt, such that the same sequence of instructions,

but with a modified latency for the first instruction $\tau'_\pi(i_1) = \tau_\pi(i_1) + \Delta t$, results in a different sequence of instruction latencies $\tau_\pi(i_1) + \Delta t : \tau_\pi(i_2) : \cdots : \tau_\pi(i_n)$ and thus a possible different total execution time C', and thus a timing difference of $\Delta C = C' - C$.

Definition 60 (Timing Anomalies by Changes in Instruction Latency [98])**.** *A timing anomaly is defined as a situation where according to the sign of* Δt *one of the following cases become true:*

a) Increase of the latency: $\Delta t > 0 \implies (\Delta C > \Delta t) \vee (\Delta C < 0)$

b) Decrease of the latency: $\Delta t < 0 \implies (\Delta C < \Delta t) \vee (\Delta C > 0)$

This definition has some drawbacks:

- As pointed out in [89] there is an underlying assumption that the latency change of the first instruction does not influence the latencies of the subsequent instructions. This is not always the case.
- In [75] the change in latency can be unrelated to a change in hardware state, resulting in the definition deeming a hardware platform to suffer from timing anomalies, while the actual platform does not.

In [98] the second point is alleviated as the change in latency is assumed to be associated to two different initial hardware states, which are further assumed to be "almost identical". However without a formalisation of "almost identical" it could be argued that the two LRU caches in Figure 8.2 are almost identical, and thus would be deemed timing anomalous.

8.4.2 Defining Timing Anomalies by Abstract Models

In [89] a formal definition of timing anomalies is given. It however states that "Non-determinism – which is necessary for timing anomalies – is only introduced by abstraction", and goes on to define timing anomalies in terms of non-deterministic hardware models. We note that timing anomalies as demonstrated originally in [75] do not involve non-determinism, but instead involve concrete traces run on the same concrete (deterministic) hardware model.

We will argue that non-determinism is *not* necessary for timing anomalies to occur. Indeed, there is a strong link between the presence of timing anomalies and the existence of a sound deterministic over-approximating model of the timing of the hardware, but the causality is not both ways. In some cases timing anomalies can even occur as artefacts of the abstraction, even though they are not present in the concrete hardware.

As an example consider the two traces in Figure 8.3. Here different input data results in very different instruction streams, sharing similarities with timing anomalous traces. In our opinion this should not be characterised as

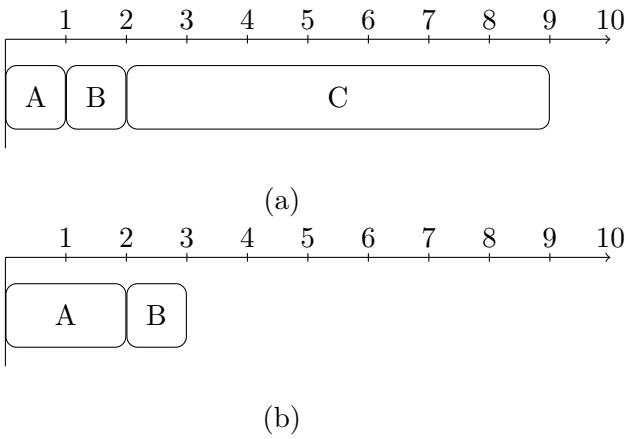


Figure 8.3: Example program run on M_3. The program is `A: MUL` R_0, R_0, R_1; `B: BRZ` R_0, C, where C is a linear, data-independent, subprogram summarised into one instruction. The instruction `BRZ` is interpreted as "branch to C if R_0 is zero". The two traces are (a) where $R_0 = 0$, and (b) where $R_0 = 1$.

a timing anomaly, as this would render practically all programs accepting input on all platforms to be timing anomalous. The definition in [89] requires that the two traces must have the same instruction streams, and thus Figure 8.3 is not a timing anomaly by that definition.

We however note that the same instruction stream can still result in two very different timing behaviours, when given different input data. As an example, the ARM architecture allows the conditional execution of most instructions. We can thus derive an example where the same instruction stream gives rise to timing anomalous behaviour on different input data, as seen in Figure 8.4.

According to the definition in [89] this would be timing anomalous, as there exists a non-local worst-case path (the A instruction in (a)) resulting in a globally longer path, than all local worst-case paths (b).

In [88] a slightly relaxed definition from [89] is given, which is used to compute upperbounds on the the possible error in WCET estimation between two hardware states. However, with regards to the examples we consider there is no difference.

In the same line [48] describes a method to identify timing anomalies in a processor using bounded model checking. This is done by comparing the execution time of the same instruction stream on two different processors: the real processor, and an (abstract) "always-worst case" performing processor. If the "worst-case" processor can overtake the real processor, the processor is deemed to have timing anomalies.

However [48] uses abstraction of input data and thus ends up comparing

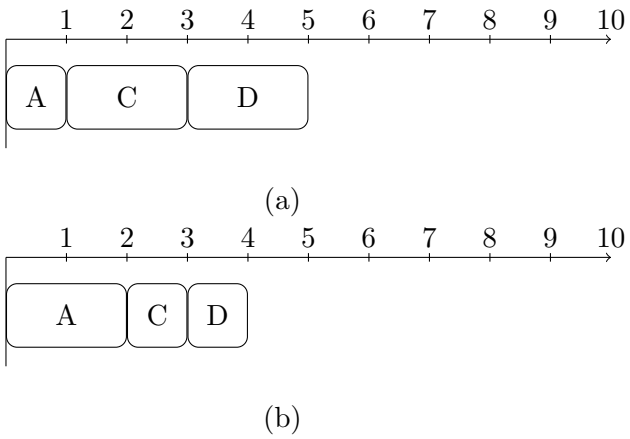


Figure 8.4: The example from Figure 8.3, but instead of branching it uses conditional execution. Therefore the two instruction streams are the same, but the processor treats C as a no-op in (b). The program is `A: MUL` R_0, R_0, R_1; `C: MULNZ` R_2, R_2, R_1; `D: MULNZ` R_2, R_2, R_1 , where we let A set the condition flags. Thus C and D only gets executed if R_0 is not 0.

execution traces which can only result from different input data: The trace given in [48] (a) cannot occur on the real processor with the same input data as the trace in (b). For trace (a) to occur one of the operands (R_4 and R_6 in this case) needs to be 0, that is $R_4 = 0 \vee R_6 = 0$. However for trace (b) to occur none of the operands can be 0, thus $R_4 \neq 0 \wedge R_6 \neq 0$. As these two conditions are the negation of one another, the two traces cannot occur with the same input data. Since the two traces cannot occur with the same input data, they will be incomparable, per our Definition 56. Of course, hardware can be viewed abstractly. For timing analysis it is very important that these abstractions are sound, i.e. overapproximating the timing.

Definition 61 (More Favorable Hardware)**.** *Hardware $\mathcal{H}$ with hardware states* $\mathsf{HardwareStates}$ *is more favorable than hardware $\mathcal{H}'$ with hardware states* $\mathsf{HardwareStates}'$*, written $\mathcal{H} \sqsubseteq \mathcal{H}'$, if there exists a mapping*

$$\alpha : \mathsf{HardwareStates} \rightarrow \mathsf{HardwareStates}'$$

s.t. $\forall P, \forall d \in \mathsf{Data}(P) : h \sqsubseteq_{time} \alpha(h)$, where $\sqsubseteq_{time}$ is extended across different hardware systems.

Typically α is an abstraction function. Clearly, if $\mathcal{H} \sqsubseteq \mathcal{H}'$, then for any program P, $WCET_{\mathcal{H}}(P) \leq WCET_{\mathcal{H}'}(P)$. $\mathcal{H}'$ is thus a sound abstraction for computing the WCET of any program. The technique presented in [48] is a very valuable tool in finding sound abstractions, however, the unsoundness of an abstraction cannot be translated into the real hardware exhibiting timing anomalies.

8.5 Results

We will now compare the different definitions of timing anomalies, and how they hold for and apply to different examples:

Lemma 13. *The classic example in Figure 8.1 is timing anomalous by Definition 57.*

Proof. We will show that none of the two initial hardware states is consistently worse than the other, per Definition 56, and thus no upper bound can exist, per Definition 57. The only two initial hardware states that are relevant to consider is the cache where the data item referenced by A is in the cache, and a state where the data item referenced by A is not in the cache. Since there are only two initial hardware states, one or both of them would have to be consistently slower than the other. In Example 4 we already showed that none of the two traces is consistently slower than the other. Therefore, Definition 57 cannot be fulfilled. □

Lemma 14. *The control-flow example in Figure 8.3 is not timing anomalous by Definition 57.*

Proof. Since M_3 has no cache, there is actually only one initial hardware state, h_0, where the first instruction is able to enter the processor in the first cycle: the empty pipeline. For every program P and data d there is therefore only one element in $I(P, d)$. By Definition 57 and the reflexivity of $\sqsubseteq_{\text{time}}$ the hardware system is timing anomaly free. □

Lemma 15. *The "branching by conditional execution" example in Figure 8.4 is not timing anomalous by Definition 57.*

Proof. The argumentation is the same as for Lemma 14. □

Lemma 16. *LRU caches are not timing anomalous by our Definition 58.*

Proof. A stronger statement can actually be proven: that the empty cache is always the worst initial hardware state for LRU caches [87]. By Definition 57 this satisfies Definition 58. □

Lemma 17. *FIFO caches are timing anomalous by our Definition 58.*

Proof. Consider the two traces in Figure 8.5, none of which are consistently slower than the other. By Lemma 12 an upper bound should exist. An upper bound would have to have misses for all accesses. By enumeration of all distinct initial caches, none of them have misses for all accesses, and thus no upper bound for the two traces exist. □

	d e		b a	
a	a d	x	b a	
c	c a	x	c b	x
a	c a		a c	x
b	b c	x	b a	x
c	b c		c b	x
a	a b	x	a c	x
b	a b		b a	x
c	c a	x	c b	x

Figure 8.5: Example of a timing anomaly for the FIFO cache on M_1, adopted from [21]. The first line is the initial state of the cache for the two traces. The first column is the access sequence, and the x'es indicate cache misses.

8.6 Conclusion and Future Work

In this work we have looked at previous definitions of timing anomalies, and identified flaws in them. Specifically in their applicability to various types of known timing anomalies, but also in what examples they deem to be timing anomalies. We have proposed a definition of timing anomalies in terms of the existence of a consistently worst initial hardware state, in the concrete model of the hardware and shown that it coincides with common knowledge about timing anomalies.

The next step is to provide an operational definition of timing anomalies that enables us to effectively check whether some hardware is timing anomalous, and if it is, identify a set of initial hardware states, such that they are consistently worse than all other hardware states. This would enable a WCET analysis by simulating the execution of these initial states. The framework we have proposed can also be used to take advantage of the efficient abstraction techniques to over-approximate WCET on timing anomalous platforms: given $\mathcal{H}$ which is timing anomalous, define $\mathcal{H}'$ that soundly approximates $\mathcal{H}$ and show that $\mathcal{H}'$ is timing anomaly free.

	Our Definitions 57, 58	Latency change [75, 98]	Abstraction [48, 89, 88]
Classic Ex. [75] (Fig. 8.1)	Yes, Lemma 13	Yes	Yes
LRU Cache	No, Lemma 16	Inapplicable[1]	No
FIFO Cache	Yes, Lemma 17	Inapplicable[1]	Yes[4]
Branching (Fig. 8.3)	No, Lemma 14	Inapplicable[2]	No[4]
Conditional exec. (Fig. 8.4)	No, Lemma 15	Yes	Yes[4]
MUL 0-speedup [48, Fig. 3]	No	Yes[3]	Yes[4]

[1] Because the latency change can in general not be limited to, or contained within, the first instruction.
[2] Because the sequence of instructions are not the same.
[3] If we allow the latency change to occur on the second instruction.
[4] Depends on the abstraction.

Chapter 9

Conclusion

Abstract interpretation and model checking are converging at a practical level. In this thesis the formalism of lattice automata has been presented. It has been shown how lattice automata can be model checked, and how abstract interpretation can be used to extract lattice automata from programs. This methodology results in a program model that is directly traceable back to the program semantics, but that still allows the program analyst to explore the model and perform ad hoc experiments, such as modelling parts of the program interaction that an abstract interpretation cannot capture — such as combining lattice automata extracted from different languages. The method of trace partitioning has been shown to be a model transformation in this framework, allowing to tune the precision and cost of the analysis.

A multi-core model checking toolchain has been presented, making efficient and scalable fixpoint computation over lattice automata possible. The toolchain has been shown scalable up to 48 cores, reducing computation times of hours to minutes and seconds.

Finally, the problem of obtaining models soundly overapproximating a hardware platform has been explored, and a definition of timing anomalies has been presented. Importantly, the definition does not rely on any abstraction itself.

It is the hope that readers of this thesis will be able to use the concepts and terminology to more closely understand the relationship between program analysis, abstract interpretation and model checking — on points of similarity and difference, likewise. Or at the very least, that the reader will be confused on a higher level.

Bibliography

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General Decidability Theorems for Infinite-State Systems. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 313–321, jul 1996.

[2] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[3] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of the Third International Symposium on Static Analysis (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66, London, UK, 1995. Springer-Verlag.

[4] R. Alur. Timed Automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 688–688, Trento, Italy, 1999. Springer.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[6] T. Amnell, G. Behrmann, J. Bengtsson, P. D'argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. Larsen, M. Möller, et al. UPPAAL - Now, next, and future. In *4th Summer School on Modeling and verification of parallel processes (MOVEP)*, volume 2067 of *Lecture Notes in Computer Science*, pages 99–124. Springer, 2001.

[7] ARM Limited. *ARM920T Technical Reference Manual*, 1. edition, 2001.

[8] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening Operators for Powerset Domains. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*,

volume 2937 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2004.

[9] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[10] T. Ball and S. Rajamani. The SLAM toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, 2001. Springer.

[11] J. Barnat and P. Rockai. Shared Hash Tables in Parallel Model Checking. In *Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, volume 198 of *Electronic Notes in Theoretical Computer Science*, pages 79–91, 2007.

[12] G. Behrmann. Distributed reachability analysis in timed automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(1):19–30, 2005.

[13] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL Implementation Secrets. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRFT)*, volume 2469 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2002.

[14] G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static Guard Analysis in Timed Automata Verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 254–277. Springer, 2003.

[15] G. Behrmann, P. Bouyer, K. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 2004.

[16] G. Behrmann, A. David, and K. Larsen. A Tutorial on UPPAAL. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems, Formal Methods for the Design of Real-Time Systems (SFM-RT)*, volume 3185 of *Lecture Notes in Computer Science*, pages 33–35, Bertinoro, Italy, 2004. Springer.

[17] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Developing Uppaal over 15 years. *Software: Practice and Experience*, 41(2):133–142, February 2011.

[18] G. Behrmann, T. Hune, and F. W. Vaandrager. Distributing Timed Model Checking - How the Search Order Matters. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2000.

[19] J. Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, 2002.

[20] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2004.

[21] C. Berg. PLRU Cache Domino Effects. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[22] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518, Berlin, Germany, 2007. Springer.

[23] S. Biallas, M. C. Olesen, F. Cassez, and R. Huuck. PtrTracker: Pragmatic Pointer Analysis. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, (to appear). IEEE Computer Society, 2013.

[24] S. Blom, J. C. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359, Edinburgh, UK, 2010. Springer.

[25] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-Fly Symbolic Model Checking for Real-Time Systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 25–34. IEEE Computer Society, 1997.

[26] P. Bouyer. Untameable Timed Automata! In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2607 of *Lecture Notes in Computer Science*, pages 620–631, Berlin, Germany, 2003. Springer.

[27] P. Bouyer. Forward Analysis of Updatable Timed Automata. *Formal Methods in System Design*, 24(3):281–320, 2004.

[28] V. A. Braberman, A. Olivero, and F. Schapachnik. Dealing with Practical Limitations of Distributed Timed Model checking for Timed Automata. *Formal Methods in System Design*, 29(2):197–214, 2006.

[29] J. Brauer, R. R. Hansen, S. Kowalewski, K. G. Larsen, and M. C. Olesen. Adaptable Value-Set Analysis for Low-Level Code. In *Proceedings of the 6th International Workshop on Systems Software Verification (SSV)*, pages 32–43, 2011.

[30] J.-L. Béchennec and F. Cassez. Computation of WCET using Program Slicing and Real-Time Model-Checking. Research Report, IRCCyN/CNRS, May 2011.

[31] F. Cassez, R. R. Hansen, and M. C. Olesen. What is a Timing Anomaly? In T. Vardanega, editor, *Proceedings of the 12th International Workshop on Worst-Case Execution-Time Analysis (WCET)*, OASIS, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[32] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[33] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, 1992.

[34] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[35] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, San Antonio, Texas, USA, 1979. ACM Press.

[36] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(1):103–179, 1992.

[37] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Leuven, Belgium, 1992. Springer.

[38] P. Cousot and R. Cousot. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[39] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proceedings of the 14th European Symposium on Programming, Languages and Systems (ESOP)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, UK, 2005. Springer.

[40] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does ASTRÉE scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[41] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–96, Tucson, Arizona, USA, 1978. ACM Press.

[42] A. E. Dalsgaard, R. R. Hansen, K. Y. Jørgensen, K. G. Larsen, M. C. Olesen, P. Olsen, and J. Srba. opaal: A Lattice Model Checker. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *Proceedings of the International Symposium NASA Formal Methods (NFM)*, volume 6617 of *Lecture Notes in Computer Science*, pages 487–493. Springer, 2011.

[43] A. E. Dalsgaard, A. Laarman, K. G. Larsen, M. C. Olesen, and J. v. d. Pol. Multi-core Reachability for Timed Automata. In *Proceedings of the 11th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 7595 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2012.

[44] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis Using Model Checking. In B. Lisper, editor, *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 114–124, 2010.

[45] C. Daws and S. Tripakis. Model Checking of Real-Time Reachability Properties Using Abstractions. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 313–329, Lisbon, Portugal, 1998. Springer.

[46] M. del Mar Gallardo, J. Martinez, P. Merino, and E. Pimentel. aSPIN: Extending SPIN with abstraction. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, volume 2318 of

Lecture Notes in Computer Science, pages 241–252, Grenoble, France, 2002. Springer.

[47] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212, Grenoble, France, 1990. Springer.

[48] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, pages 13–18, 2006.

[49] S. Evangelista, A. W. Laarman, L. Petrucci, and J. C. v. d. Pol. Improved Multi-Core Nested Depth-First Search. In S. Ramesh, editor, *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 7561 of *Lecture Notes in Computer Science*, pages 269–283, Kerala, India, 2012. Springer.

[50] S. Evangelista, L. Petrucci, and S. Youcef. Parallel Nested Depth-First Searches for LTL Model Checking. In *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 6996 of *Lecture Notes in Computer Science*, pages 381–396, Taipei, Taiwan, 2011. Springer.

[51] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[52] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, 1982.

[53] G. Gebhard. Timing Anomalies Reloaded. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 15, pages 1–10. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[54] G. Geeraerts, J.-F. Raskin, and L. Van Begin. Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. *Journal of Computer and System Sciences*, 72(1):180, 2006.

[55] D. Gopan and T. Reps. Guided static analysis. In *Proceedings of the 14th International Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 349–365, Kongens Lyngby, Denmark, 2007. Springer.

[56] M. Handjieva and S. Tzolovski. Refining Static Analyses by Trace-Based Partitioning Using Control Flow. In *Proceedings of the 5th International Static Analysis Symposium (SAS)*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 1998.

[57] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 2–13. IEEE, 1997.

[58] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[59] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.

[60] G. J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In *Proceedings of the Second SPIN Workshop (SPIN)*, volume 32, pages 23–32. American Mathematical Society, 1996.

[61] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic Partitioning in Analyses of Numerical Properties. In *Proceedings of the 6th International Symposium Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 39–50, Venice, Italy, 1999.

[62] B. Jeannet and A. Miné. APRON: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667, Grenoble, France, 2009. Springer.

[63] T. Jensen, H. Pedersen, M. C. Olesen, and R. R. Hansen. THAPS: Automated Vulnerability Scanning of PHP Applications. In *Proceedings of the 17th Nordic Conference on Secure IT Systems (NordSec)*, volume 7617 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2012.

[64] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[65] O. Kupferman and Y. Lustig. Lattice Automata. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 4349 of *Lecture Notes in Computer Science*, pages 199–213, Nice, France, 2007. Springer.

[66] A. Laarman, M. C. Olesen, A. Dalsgaard, K. G. Larsen, and J. van de Pol. Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 968–983, Saint Petersburg, Russia, 2013. Springer.

[67] A. Laarman and J. v. d. Pol. Variations on Multi-Core Nested Depth-First Search. In *Proceedings of the 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, volume 72, pages 13–28. EPTCS, 2011.

[68] A. Laarman, J. v. d. Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *Proceedings of the International Symposium NASA Formal Methods (NFM)*, volume 6617 of *Lecture Notes in Computer Science*, pages 506–511, 2011.

[69] A. Laarman, J. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In N. Sharygina and R. Bloem, editors, *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Swiss*, USA, October 2010. IEEE Computer Society.

[70] A. Laarman, J. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In *Proceedings of the 18th International SPIN Workshop on Model Checking Software*, volume 6823 of *Lecture Notes in Computer Science*, pages 38–56, Snowbird, UT, USA, 2011. Springer.

[71] A. W. Laarman, R. Langerak, J. C. v. d. Pol, M. Weber, and A. Wijs. Multi-Core Nested Depth-First Search. In T. Bultan and P. A. Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 6996 of *Lecture Notes in Computer Science*, Tapei, Taiwan, July 2011. Springer.

[72] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[73] G. Li. Checking timed Büchi automata emptiness using LU-abstractions. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 5813 of *Lecture Notes in Computer Science*, pages 228–242, Budapest, Hungary, 2009. Springer.

[74] T. Lundqvist. *A WCET analysis method for pipelined microprocessors with cache memories*. PhD thesis, Chalmers University of Technology, 2002.

[75] T. Lundqvist and P. Stenstrom. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, Phoenix, AZ, USA, 1999. IEEE Computer Society.

[76] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Proceedings of the 14th European Symposium on Programming Languages and Systems (ESOP)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20, Edinburgh, UK, 2005. Springer.

[77] A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Proceedings of the Second Symposium on Programs as Data Objects (PADO)*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, May 2001. http://www.di.ens.fr/mine/publi/article-mine-padoII.pdf.

[78] A. Miné. The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*, pages 310–319, Stuttgart, Germany, Oct. 2001. IEEE Computer Society.

[79] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006.

[80] D. Monniaux. The parallel implementation of the Astrée static analyzer. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 2005.

[81] F. Nielson and H. R. Nielson. Model Checking Is Static Analysis of Modal Logic. In L. Ong, editor, *Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS)*, volume 6014 of *Lecture Notes in Computer Science*, pages 191–205. Springer-Verlag, 2010.

[82] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[83] M. C. Olesen, R. R. Hansen, and K. G. Larsen. An Automata-Based Approach to Trace Partitioned Abstract Interpretation. 2013.

[84] M. C. Olesen, R. R. Hansen, J. Lawall, and N. Palix. Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Coding Standard certification. In *Pre-proceedings of the 4th International*

Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert), volume 33 of *Electronic Communications of the EASST*, pages 51–69, 2010.

[85] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix. Coccinelle: Tool support for automated CERT C Secure Coding Standard certification. *Science of Computer Programming (SCP)*, 2012.

[86] P. Olsen, K. G. Larsen, and A. Skou. Present and Absent Sets: Abstraction for Testing of Reactive Systems with Databases. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 264(3):53–68, 2010.

[87] J. Reineke and D. Grund. Sensitivity of Cache Replacement Policies. Technical Report 36, AVACS (Automatic Verification and Analysis of Complex Systems), March 2008. ISSN: 1860-9821, http://www.avacs.org/.

[88] J. Reineke and R. Sen. Sound and Efficient WCET Analysis in the Presence of Timing Anomalies. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 10 of *OASICS*, page 101, Dublin, Ireland, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[89] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[90] X. Rival and L. Mauborgne. The Trace Partitioning Abstract Domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.

[91] P. Sanders. Lastverteilungsalgorithmen fur Parallele Tiefensuche. number 463. In *in Fortschrittsberichte, Reihe 10. VDI.* Verlag, 1997.

[92] D. A. Schmidt and B. Steffen. Program Analysis as Model Checking of Abstract Interpretations. In *Proceedings of the 5th International Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380, Pisa, Italy, 1998. Springer.

[93] S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *In Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190, Edinburgh, UK, 2005. Springer.

[94] S. Tripakis. Checking Timed Büchi Automata Emptiness on Simulation Graphs. *ACM Transactions on Computational Logic (TOCL)*, 10(3):15, 2009.

[95] S. Tripakis, S. Yovine, and A. Bouajjani. Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.

[96] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the 1st Symposium on Logic in Computer Science (LICS)*, pages 332–344, Cambridge, June 1986.

[97] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[98] I. Wenzel, R. Kirner, P. P. Puschner, and B. Rieder. Principles of Timing Anomalies in Superscalar Processors. In *Proceedings of the Fifth International Conference on Quality Software (QSIC)*, pages 295–306, Melbourne, Australia, 2005. IEEE Computer Society.

[99] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.